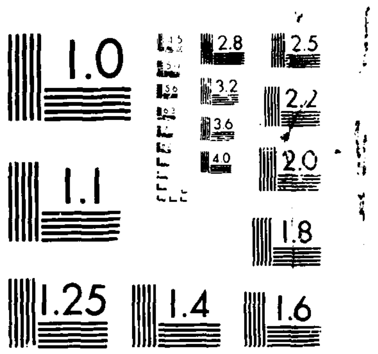
AD-A195 395 ACCESS: A COMMUNICATING AND COOPERATING EXPERT SYSTEMS 1/2
SYSTEM(U) SYMBIOTICS INC CAMBRIDGE MA
B H COTTMAN ET AL. 31 JAN 88 DAAB10-87-C-0053
UNCLASSIFIED F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(1)

Final Report

ACCESS
A Communicating and Cooperating Expert Systems System

Topic No: SBIR A87-319, Department of the Army.

Name and Address of Contracting Small Business Firm
Symbiotics Inc.
875 Main Street
Cambridge Ma 02139-3909

Phase 1 Contract: DAAB10-87-C-0053, U.S. Army Signals Warfare Center.
Contract Effective Date: 30-June-1987
Contract Expiration Date: 30-December-1987
Reporting Period: 30-June-1987 to 31-January-1988

Name and Title of Principal Investigators
Dr. Bruce H. Cottman, Vice President of Engineering
Dr. Robert C. Paslay, Vice President of Technology
(617)876-3635

Project Title
ACCESS: A Communicating and Cooperating Expert Systems System

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the offical policies, either expressed or implied, of the Government.

ACCESS
A Communicating and Cooperating Expert Systems System

The primary focus of Phase I was to prototype a development environment, **ACCESS**, for A Communicating and Cooperating Expert Systems System. More generally, this work explored the question of what capabilities were needed in a development environment for embedding distributed knowledge-based systems applications on personal computer or work-station class platforms. The stated goal of the Phase I research and development effort was to investigate and implement a software environment for the realization of cooperating knowledge sources on personnal computers. This system was to be Lisp based, distributed processing was to be facilitated by message passing using TCP/IP, control was to be accomplished by meta-level objects and a variety of features were to be provided to aid developers in building such systems. Underlying these goals was the assumption that the tools needed to support such an effort, mainly Common Lisp, Portable Common Loops and TCP/IP, were adequate to do so. During the course of this work **Symbiotics** found several shortcomings in these software tools and identified a need for higher level tools to facilitate distributed processing development. This report documents that work and the results of the Phase I effort.

Symbiotics designed and implemented a language based environment to allow the user to develop and manipulate the distributed processing aspects of the **ACCESS** system. Specifically, as part of the Phase I effort, **Symbiotics** chose to develop a high level language to facilitate the distribution of information to problem solving agents on separate processors. This language, **ORGAL**, provides a comprehensive environment for distributed processing development on a variety of software and hardware architectures. **ORGAL** has the desired qualities of being interactive, extensible, and intuitive. It allows the user to develop heterogeneous distributed processing applications using an unique model for asynchronous message passing which embodies control within the messages themselves. In this manner, the user can distribute their application quickly and interactively while achieving maximum computational efficiency. **ORGAL** is currently being ported to a variety of Lisp environments as well as to the language C in anticipation of commercial release in the fourth quarter of 1988.

Another finding of the Phase I effort addresses the current state of affairs of commercial knowledge-based system development tools. These tools were developed for stand-alone or centralized processing and are not ideally suited to cooperating knowledge-based agent applications. This is largely because the developers of these systems envisioned users embedding their application in the shells themselves. Distributed processing utilizing knowledge-based systems is better served by "artificial intelligence" functionality which can be embed-

A195325

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704-0188 Exp Date Jun 30, 1986
1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): –	5. MONITORING ORGANIZATION REPORT NUMBER(S): –
6a. NAME OF PERFORMING ORGANIZATION: SYMBIOTICS INC. / 6b. OFFICE SYMBOL (If applicable): –	7a. NAME OF MONITORING ORGANIZATION: U.S. ARMY CENTER FOR SIGNALS WARFARE
6c. ADDRESS (City, State, and ZIP Code): 875 MAIN STREET CAMBRIDGE MA 02139-3909	7b. ADDRESS (City, State, and ZIP Code): VINT HILL FARMS STATION WARRENTON, VA 22186
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: USACSW / 8b. OFFICE SYMBOL (If applicable): AMSEL-RD-SW-PC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAB10-87-C-0053
8c. ADDRESS (City, State, and ZIP Code): VINT HILL FARMS STATION WARRENTON, VA 22186	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
ACCESS - A COMMUNICATING AND COOPERATING EXPERT SYSTEMS SYSTEM

12. PERSONAL AUTHOR(S)
DR. BRUCE H. COTTMAN, DR. ROBERT C. PASLAY

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
FINAL	FROM 30 JUNE 87 TO 31 JAN 88	88 01 31	110

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	COOPERATING EXPERT SYSTEMS, DISTRIBUTED AI, CONCURRENT PROCESSES, COMMUNICATION TOPOLOGIES, PARALLEL PROCESSING, COMMON LISP, ORGANIZATIONAL MODELS.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The primary focus of Phase I was to prototype a development environment, ACCESS, for A Communicating and Cooperating Expert Systems System. More generally, this work explored the question of what capabilities were needed in a development environment for embedding distributed knowledge-based systems applications on personal computer or work-station class platforms. The stated goal of the Phase I research and development effort was to investigate and implement a software environment for the realization of cooperating knowledge sources on personnal computers. This system was to be Lisp based, distributed processing was to be facilitated by message passing using TCP/IP, control was to be accomplished by meta-level objects and a variety of features were to be provided to aid developers in building such systems. Underlying these goals was the assumption that the tools needed to support such an effort, mainly Common Lisp, Portable Common Loops and TCP/IP,

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL: CHRIS BOGART	22b. TELEPHONE (Include Area Code): (703) 347-6438 / 22c. OFFICE SYMBOL: AMSEL RD-SW-PC

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

19.

were adequate to do so. During the course of this work **Symbiotics** found several shortcomings in these software tools and identified a need for higher level tools to facilitate distributed processing development. This report documents that work and the results of the Phase I effort.

ded in the application. This observation has lead to the development of **KNO**, an object oriented, open architecture package for knowledge-based system development. Foremost in the design effort of **KNO** was to realize the development of a set of abstractions that would allow the application of specialized A.I. technologies on the basis of the structure and the function of the knowledge representated. The goal was to realize an architecture that would allow the functionality of various reasoning technologies typified by Truth Maintenance Systems, backward and forward chaining inference engines, and various representation technologies typified by semantic nets. Krypton, rule-based, logic-based and framed-based to be commonly representable as a coherent whole. During the design of the **KNO** operation protocols, the subsequent development of the **KNO** rule-based representation compiler, and the development of **ORGAL** it was realized that a language-based architecture coupled with specialized compilers could be the basis on which **KNO** would be able to incorporate newly developed or existing artificial intelligence technologies or tools by description instead of the more costly method of reimplementating.

The result of the Phase I effort is that **Symbiotics** prototyped a general distributed processing environment for a heterogeneous collection of hardware and software platforms. It is these findings which lay the foundation for our future work. In particular, the fundamental question that was resolved was, *What minimal set of generic functionality must be provided by a development system to be able to represent an organization of problem solving agents?*. The Phase I effort has resulted in **Symbiotics** identifying major issues in distributed processing as well as making a significant contribution to resolving these issues.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The primary focus of Phase I was to prototype a development environment, **ACCESS**, for A Communicating and Cooperating Expert Systems System. More generally, this work explored the question of what capabilities were needed in a development environment for embedding distributed knowledge-based systems applications on personal computer or workstation class platforms. The stated goal of the Phase I research and development effort was to investigate and implement a software environment for the realization of cooperating knowledge sources on personnal computers. This system was to be Lisp based, distributed processing was to be facilitated by message passing using TCP/IP, control was to be accomplished by meta-level objects and a variety of features were to be provided to aid developers in building such systems. Underlying these goals was the assumption that the tools needed to support such an effort, mainly Common Lisp, Portable Common Loops and TCP/IP, were adequate to do so. During the course of this work **Symbiotics** found several shortcomings in these software tools and identified a need for higher level tools to facilitate distributed processing development. This report documents that work and the results of the Phase I effort. [illegible]

Symbiotics designed and implemented a language based environment to allow the user to develop and manipulate the distributed processing aspects of the **ACCESS** system. Specifically, as part of the Phase I effort, **Symbiotics** chose to develop a high level language to facilitate the distribution of information to problem solving agents on separate processors. This language, **ORGAL**, provides a comprehensive environment for distributed processing development on a variety of software and hardware architectures. **ORGAL** has the desired qualities of being interactive, extensible, and intuitive. It allows the user to develop heterogeneous distributed processing applications using an unique model for asynchronous message passing which embodies control within the messages themselves. In this manner, the user can distribute their application quickly and interactively while achieving maximum computational efficiency. **ORGAL** is currently being ported to a variety of Lisp environments as well as to the language C in anticipation of commercial release in the fourth quarter of 1988.

Another finding of the Phase I effort addresses the current state of affairs of commercial knowledge-based system development tools. These tools were developed for stand-alone or centralized processing and are not ideally suited to cooperating knowledge-based agent applications. This is largely because the developers of these systems envisioned users embedding their application in the shells themselves. Distributed processing utilizing knowledge-based systems is better served by "artificial intelligence" functionality which can be embedded in the application. This observation has lead to the development of **KNO**, an object oriented, open architecture package for knowledge-based system development. Foremost in the design effort of **KNO** was to realize the development of a set of abstractions that would allow the application of specialized A.I. technologies on the basis of the structure and the function of the knowledge representated. The goal was to realize an architecture that would allow the functionality of various reasoning technologies typified by Truth Maintenance Systems, backward and forward chaining inference engines, and various representation technologies typified by semantic nets, Krypton, rule-based, logic-based and framed-based to be commonly representable as a coherent whole. During the design of the **KNO** operation protocols, the subsequent development of the **KNO** rule-based representation compiler, and the development of **ORGAL** it was realized that a language-based architecture coupled with specialized compilers could be the basis on which **KNO** would be able to incorporate newly developed or existing artificial intelligence technologies or tools by description instead of the more costly method of reimplementating.

The result of the Phase I effort is that **Symbiotics** prototyped a general distributed processing environment for a heterogeneous collection of hardware and software platforms. It is these findings which lay the foundation for our future work. In particular, the fundamental question that was resolved was, *What minimal set of generic functionality must be provided by a development system to be able to represent an organization of problem solving agents?*. The Phase I effort has resulted in **Symbiotics** identifying major issues in distributed processing as well as making a significant contribution to resolving these issues.

1.1 Introduction and Specification of Problem

The ultimate goal of an intelligent system is the ability to function robustly on large complex problems. We have adopted the premise of the Actor model [He77,He85b,Ag87] that only a dynamic organization of multiple specialists or agents that are self-reflective and can cooperate through communication can demonstrate the robustness required by large complex problem domains. We have also adopted the premise that the large variety of successful human organizations indicates that the development of an organizational model is dominated by the problem domain. The forms that an organization of cooperating agents can assume covers a wide range of models. Examples of organizational models are dictatorial, where one agent is responsible for the procedural problem solving flow; delegatory, where problem solving control is distributed in a top-down hierarchical fashion; or participatory, where problem solving specialists assume subtask responsibility on the basis of mutual consent [Ba86,Be86,Da81,Ha86,Sa85,Sm85,St84]. However, whatever the organizational model, each have in common the ability to adapt to an evolving problem domain. The goal of expert systems, logic based systems and knowledge-based systems in general is to find solutions to tasks presented to them in their problem domain. It is a fundamental shortcoming of these systems that they cannot mimic the adaptive problem solving that is so often found among a team of humans beings. One solution to this problem is to have autonomous knowledge-based objects communicating and negotiating in ways analogous to human organizations [Kr81,He86].

We are currently designing a commercial product which will provide the flexibility to capture as many organizational paradigms as possible [Sy86]. In order to do this, one must abstract to a level where the characteristics which organizations display are commonly representable. To date, the abstractions identified are modes of distributed communication, description of distributed resource state, resource management, organizational structure, knowledge representation and knowledge processing.

Modes of Distributed Communication

Modes of communication can be discrete (e.g. agent A to agent B) or global (e.g. agent A to all agents). Arrival and response order can be determinate or nondeterminate. The content and intent of a communication is determined by the application requirements. To facilitate adaptation of agents to different problem domains, an intelligent system development environment must have a flexible communication scheme allowing for variable content, multiple targets and different arrival/response ordering among agents.

Description of Distributed Resource State

A system composed of intelligent agents distributed over multiple environments must be able to access the description of the state of the resources on which it relies. Resource and task management require accurate and timely discriptions of the current state of the resource enviroment. In a distributed environment, where agents are free to move or create new agents, a separate development substrate for maintaining distributed resource descriptions integrated with support services and application interfaces is required.

Resource Management
It is not uncommon for an organization to have an agenda or plan by which to proceed. These plans normally include allocation of time, money and other resources to predefined tasks. In fact, some proposed cooperating expert system schemes [Ko81,Ma85,He85a] would not be tractable if tasks were not limited in some chosen resource. Resource management is a capability that must be directly available to both the distributed application and the underlying system components which it tasks.

Organizational Structure
The structural arrangement of members in an organization determines the topology of the organization. The design decision of which organizational topology to use is the prerogative of the problem domain under study. A development environment for distributed applications must be able to accommodate a wide range of organizational topologies.

Knowledge Representation and Processing
In order to develop efficient and effective distributed artificial intelligence based applications the knowledge presentation, knowledge processing and distributed communication resources must be integrated in an uniform and systematic manner. Also, application efficiency and upward evolution can only be realized by a development environment that allows the application of specialized knowledge representation and processing on the basis of the structure and function of the knowledge.

1.2 Distributed Processing and Communication

During the past twenty years many schemes for realizing distributed processing have been proposed and a few actually implemented. To date, most of these paradigms have remained under the auspices of both academic and corporate research departments. Related commercially released products have been limited to file transfer utilities [Sp87], electronic mail packages [Lo86] and low level remote procedure call facilities [Ne81]. The following sections review previous work in the relevent areas of distributed processing and communication. the issues involved and how the Phase I effort builds on this previous work.

1.2.1 Heterogeneous Distributed Processing

Once the decision to connect two processors together has been made, the first question which must be addressed is which processors to use. If the two processors are identical, we refer to such a system as a homogeneous system. If they are different then the system is a heterogeneous system. These two different cases present different problems to the developer and thus affect the design. The vast majority of implemented systems have been done on homogeneous systems. This is largely due to the fact that the software implementation is greatly simplified when there is only one set of hardware, operating system, and communication specific issues to address. In addition, these same simplifying features are attractive from a specialized parallel hardware design standpoint. These advantages together with the locality of processors afforded by specialized hardware have greatly facilitated both synchronized processes and shared memory models. These paradigms are desirable because they are more intuitive than asynchronous message passing[Ag87] or dataflow models [Ag82] and are therefore more quickly accepted by programmers. This is in part due to the already familiar concepts of multi-processing environments (e.g. semaphores [Di68] and monitors [Br75]) which can be viewed as a special case of synchronized shared memory models [Di68].

Heterogeneous systems however, are by far the most common class configurations found in present day computing facilities. While several authors [Li79,Br75,Ly81,An86,Ag87,Ho78] claim that their models of distributed computation are amenable to heterogeneous systems. these systems are ill-suited for these environments. While shared memory can be useful for certain applications on particular heterogeneous configurations [Po87], grossly inefficient computation results in situations where locality of nodes is not possible and frequent access of memory is required. Synchronization of processes also suffers from a loss of efficiency in the absence of locality [Po87].

The pragmatics of supporting some of these systems on a wide variety of hardware architectures is also questionable. The Actor model of computation requires an infinite memory store [Ag87], PPPP [PP87] and Concurrent Pascal [Br75] require a multi-processing operating system, LOCUS [Po85] relies on the UNIX operating system.

ORGAL, developed during the Phase I effort of this proposal, may have been the first completed environment to use compiler technology to accommodate hardware differences for distributed processing [Fa87]. In doing so, it is arguably the first successful attempt at designing and implementing a distributed processing environment which promises to be truly hardware independent and thus suitable for general heterogeneous distributed processing.

1.2.2 Shared Memory and Message Passing

At the heart of any distributed processing system is a methodology for exchanging information between processes. Having more than one processor requires a physical connection be established between these processors. The only way to transmit information across this connection is by translating that information into signals which can traverse the connecting media. At the receiving end of such a transmission those signals must be collected and translated into a form usable by the receiving process. If we refer to such packets of encoded information as "messages", then all distributed processing systems utilize message passing at some level.

Following a similar line of reasoning, a sequential processor can only execute a single instruction at any given time. This means that, during the process of translation, each bit of information must be stored in a register until the entire message is translated. If we consider such registers to constitute a form of memory, then this memory can be viewed as being "shared" between the two communicating processes (i.e. one process writes to it and the other process reads from it). Consequently, all distributed processing systems also utilize a form of shared memory at some level.

Historically, researchers have chosen to build their models of distributed processing based on abstraction of one of the above underlying mechanisms. This has proven to be a fruitful approach leading to a variety of different models based on such abstractions. While the nature of this proposal is to use an even higher level of abstraction to obscure this level of detail, it is instructive to briefly describe these two approaches.

Shared Memory Model

Shared memory models [Ly81,Gr78,Br75] are useful in applications where locality of nodes is achievable and memory access traffic is light. They are appealing largely because they are easily understood and utilized by programmers with conventional computer science backgrounds. This is partly due to familiarity with the concept of shared memory in multi-processing environments. As one might expect, this paradigm of computation has an extensive history dating back almost thirty years in the literature.

The general concept is that memory is allocated as a common store for some number of processes. These processes may write or read this memory. To avoid conflicts arising from simultaneous access of this memory, synchronization of these accesses must be assured. There are a number of mechanisms for guaranteeing the sequentiality of these transactions [Mo85,Ho78]. Specialized hardware has been devised to efficiently facilitate these transactions [Po87]. In general such architectures provide the locality of processors and fast access memory necessary to make these models viable. With some notable exceptions [Hi85], this approach to specialized hardware design has proven to be the most commercially successful [Po87].

Message Passing

The abstraction most often cited for the message passing models is based on communicating objects [Bi73,Ag87]. The user is to envision objects passing information directly back and forth to each other. These objects have taken on a variety of forms in different models as have the semantics of the act of passing or sending messages. Essentially, message passing is either synchronous [An86] or asynchronous [Ag87] and the objects involved are either history sensitive [Ag87] or not [Po87]. The consequences of these choices have profound effects on the behavior of the organizations formed by the message passing objects.

Synchronized message passing requires a high degree of cooperation between objects as one object must be ready to receive when another object is ready to send. In asynchronous models, messages may be sent and received at will requiring almost no cooperation between objects. If the objects involved can change their behavior based on the messages they receive then they are history sensitive. Such systems are said to be dynamic as they can change their behavior during the course of a computation. If the objects always process the information received in a message in a predetermined way, then the system is said to be static. Dynamic asynchronous models provide the most general model of distributed computation but have proven to be difficult to program [PP87].

1.2.3 Asynchronous Processing

Maximum efficiency in a distributed computation often requires that the distributed processes proceed in an asynchronous manner. This model of computation avoids one process needlessly waiting for another to complete. While this can be easily proved it has not been readily accepted by the computing community as a high level programming paradigm. This uneasiness with computations which proceed in a non-deterministic manner can be attributed to the failure of existing models to embody control mechanisms for asynchronous processing which accomodate the users perception of what a program is and does. Fundamentally, users tend to think in terms of programmatic behavior being completely in their control a priori and are uncomfortable with the notion of computations which are not predicatable at the "line of code" level.

While it is unrealistic to anticipate asynchronous behavior at any given point in time in a heterogeneous environment with unpredictable loads on the processors involved, it is entirely possible to develop a model for asynchronous computation which allows the user to have explicit control of such computations within the computation itself. One does not know the behavior of such a system prior to running it, but one could specify that behavior programmatically to dynamicly control the computation while it is running. This can only be done by allowing access to the state of a computation while the application is executing. Control decisions can then be made at runtime based on the state of the computation.

In a distributed system, the state of any given computation is directly tied to the state of the processor on which it is running. This state is not available between processors at any point in time as communication latency negates any such information passed from one to another. However, if one postulates a virtual machine and then passes that virtual machine from processor to processor then the state of the computation is preserved between processors.

This was the driving architecture behind the design of **ORGAL**. In our paradigm, a computational stack is passed from one machine to another in the form of a message. This computational history is availiable for inspection and mutation at each node which recieves it and thus gives the user complete control of the computation at runtime. Preliminary use of **ORGAL** indicates that this model of computation is readily understood and effectly used by programmers with no formal training in the issues of parallel or distributed processing. Should this observation prove to be true in the computational community at large, the promise of heterogeneous distributed processing may become a common place reality.

1.2.4 Transparent Distributed Processing

Referential transparency [Sh86] is a concept whereby the user always utilizes the same methodology to accomplish a task even though the actual execution of their request may be performed in a number of different ways. An optimizing compiler will produce different instructions based on the environment in which any given line of a program lies. This process is intentionally hidden from the programmer so that he/she can concentrate on higher level design concepts. To date, the vast majority of distributed processing environments have required the user to explicitly deal with distributed computation issues at either the message passing or shared memory level. Part of the Phase I effort was to understand and develop supporting substrates for the transparent distribution and execution of cooperating knowledge sources. This model is based on an object oriented abstraction much like the message passing models while making the methodology of information exchange totally transparent to the user. The system itself will determine whether shared memory or message passing paradigms are appropriate in the context of a given organization of knowledge sources. This allows the user to concentrate on the organizational behavior of their application and not on the mechanisms required to support the distributed configuration. It is this characteristic which distinguishes ACCESS from its predecessors.

Chapter 2

Technical Objectives

2.1 Communication Substrate Technical Objectives

The communicaton substrate provides low-level reliable delivery of message packets from one node to another on a network. For the Phase I effort this substrate was to be Ethernet cable and the Internet Protocols.

2.2 ORGAL Technical Objectives

The **ORGA**nazational Language, **ORGAL**, provides the methodology for defining and distributing computations between heterogeneous computing environments. It is designed to provide an interactive, extensible and intuitive environment for development of distributed processing applications. Technically, **ORGAL** is based on an asynchronous message passing model which uses a unique control mechanism by which the user can embed runtime control strategies. The **ORGAL** compiler makes use of delayed evaluation and object oriented database design to acheive hardware and application independence. While the ultimate objective was to provide a message passing protocol for communicating expert systems, the **ORGAL** system was intentionally designed to be a general purpose tool for distributed processing.

2.3 KNO Technical Objectives

Knowledge Objects, **KNO**, is the substrate component of **ACCESS** that offers an uniform interface in the form of generic operations on distributed knowledge-based processes. The major technical objective of the development of the **KNO** substrate is the development of a set of abstractions that will allow the application of specialized reasoning technologies on the basis of the structure and function of the knowledge. The abstraction level must be sufficient enough that the functionality of various reasoning technologies are commonly representable in one descriptive substrate.

Basic KNO differentiates itself from other high-end commercial knowledge engineering tools, such as ART [Inf87], KEE [In85] and KnowledgeCraft [CG87] as it is *designed to be embedded in applications as opposed to applications being embedded in the knowledge engineering tool.* Direct support of input-output functionality by these tools is prohibitively narrow. I/O functionality consists almost entirely of sophisicated user interface utilities that require the Artificial Intelligence development tool to be used in an interactive mode. In fact, these user interfaces can account for more than half of the product's development and maintainance costs. In order to perform basic I/O with files, databases and other processes, the application developer must patch in their own application specific I/O functions.

Basic KNO consists of standard and well-understood knowledge representation and knowledge processing technologies. A **KN**owledge **O**bject is a knowledge base which is treated as an abstract data type. The **KNO** interacts with an user or system application only through a small set of operations. This design strategy is based on the functional approach proposed by Levesque and Brachman [Ba83]. This approach requires a knowledge base to be specified functionally, ignoring how the knowledge base is implemented.

Chapter 3

Problems Encountered During Effort

The major stumbling impediment to the our Phase I effort was the development of ACCESS using available commericial software. The two major difficulties encountered were the available Common Lisp environments on personal computer class machines, and the corresponding distributed communications packages available.

3.1 Common Lisp and Personal Computers

The greatest difficulties were due to the inadequacy of purchased software to meet our development needs. This is in part due to the fact that all available required software was in a beta-test phase. More specifically, the only personal computer class Common Lisp implementation available at the beginning of this contract was Gold Hill's Beta Version 2.9 Common Lisp. As of the writing of this report the final release version 3.0 has been shipped. This version of GCLISP, in our opinion, is marginally acceptable for serious Lisp development. Nonetheless, we were able to port Lisp code developed on Lisp machines to this environment. There is cause to be optimistic about serious Lisp environments on PC class machines. During this effort PowerLisp became available as a beta-test product and we have found this to be a promising development environment. In addition, Symbiotics will be beta testing Gold Hills 32-bit version of Lisp developed for the 80386 processor. This version promises to be a significant improvement over the present 16-bit Lisp which was used during this contract.

3.2 *Porting Lessons Learned*

The initial attempt to port Symbiotics software from the Symbolic's Lisp Machine was a total failure. There were four basic reasons for this failure, only two of which could have been anticipated. The reasons were inefficient garbage collection, unoptimizing Lisp compiler, no virtural memory support and a primitive development environment. The last two elements; no virtual memory and a primitive development environment, although not advertised, were well known "features" of the Gold Hill, and to be fair other, Lisp development environments currently available for the personal computer class platforms[1]. Anticipating this, Symbiotics had planned during the Phase I exploratory effort, to simply move over code that it had previously developed in the Symbolic's Lisp Machine environment [Sy86]. This code consisted of FBI, which was to provide the rule based knowledge representation

[1]The exception to this is a Lisp available for 80286 and 80386 class machines. *PowerLisp*, available from MicroProducts, has virtual memory support upto 30 Megabytes.

and forward and backward inference knowledge processing capability, a Prolog implementation, and **FMO**, a frame based representation that interfaced with FBI. The **FBI**, Prolog and **FMO** were to provide the knowledge representation and knowledge processing capability for our Phase I research. Using the port of this code as the starting point, Symbiotics planned to spend the major part of the Phase I effort on the research and development of the *Communicating Expert System Message Passing Protocol*, **CESMPP**, and the high level specification of knowledge-based agent behaviors such as the *Receptionist*, *Manager* and *Sponsor* [Sy86]. During the Phase I effort the **CESMPP** evolved into the development of **ORGAL**.

Our software made heavy use of the object oriented substrate offered by the Symbolics Lisp Machine, namely Flavors. We had anticipated this and had proposed porting and optimizing CommonLoops to the PC environment as a comparable replacement to the object oriented functionality offered by Flavors. The first stage of the port effort was to optimize CommonLoops [Bo86,Bo87] for speed for the Gold Hill Common Lisp environment.[2] It was during this effort, that it became apparent that the Gold Hill compiler was less than expected and the garbage collector was less than optimal. During the attempt to optimize CommonLoops, we became aware of several characteristics of the Gold Hill Lisp compiler that, since we were using a beta version, we thought were bugs. We at this point stopped our CommonLoops effort and developed a Common Lisp benchmark and test suite base on the Gabriel benchmarks [3] [Ga85] and a test suite of over 100 Common Lisp functions [4]. We discovered through experimentation with these tools and subsequent discussion with Gold Hill technical developers that their advertised benchmarks were misrepresentative of the actual product. Basically, the Gold Hill benchmark results were obtained by a *simulated* compiler. The assembler code generated by their compiler for the benchmarks was further converted by human hand into a form one would realize if an optimal compiler were available. The result was that we did achieve a significant speed up in CommonLoops but that FBI and our other software was going to have to be completely rewritten in order to fit within the limits imposed by the Gold Hill Lisp compiler, address space and garbage collector.

3.3 Networking and Communications

The other significant hurdle which had to be overcome was developing a medium for interprocessor communication. While we were able to develop and use our own implementation of the Internet Protocol [St88], this proved to be too low level a mechanism to facilitate the flexible and adaptive system design which we had originally specified. While implementing **ACCESS** a need arose to develop a language for describing various potential configurations of both expert systems and prolog systems. In doing so, we developed a substrate language for message passing, **ORGAL**, upon which we could implement features peculiar to the domain of communicating knowledge sources.

[2]This work is documented in Appendix I.

[3]For results of the Gabriel benchmarks study see Appendix G.

[4]For results of the Common Lisp performance study see Appendix E.

Chapter 4

Approach and Results

A distributed processing environment which uses heterogeneous computer resources requires an entire suite of communication protocols. Each protocol in the suite has various communication responsibilites. To help organize and standardize the various protocols that can arise, the International Standards Organization (ISO) has developed a networking reference model, the Open Systems Interconnection reference model (OSI) [ISO79].

The OSI is a hierarchical model composed of seven layers. The layers, from bottom to top are Physical, Data Link, Network, Transport, Session, Presentation, Application. This model is shown in Table 41. This model will serve as reference when explaining the various components that comprise the ACCESS architecture.

Presently, the **ACCESS** system can be broken into three distinct parts:

1. IP, the Internet Protocol, written in MicroSoft C.
2. **ORGAL**, the **ORGA**national Language providing a high level message passing protocol, implemented in Gold Hill GCLISP, PowerLisp (MicroProducts), Symbolics and Franz Common Lisp.
3. **KNO**, an object oriented knowledge representation and knowledge processing substrate.

Each of these layers will be described and discussed in the following sections.

4.1 Communication Substrate

Fundamental to the success of **ACCESS** is a medium and protocol which allows processors to communicate information to each other. The initial prototype environment utilizes an IP[Cl82] implementation. While this protocol has proven sufficient for proof of concept purposes, it does not guarantee message delivery. Currently, **TCP**[Cl82] is being implemented on top of IP to provide this necessary characteristic of robust distributed processing system design. Future implementations will incorporate the **OSI** model as well.

Table 4.1: Open Systems Interconnection Model

Application	Defines network applications available to user.
Presentation	Translates format and syntax of data for interchange across different computers
Session	Allocates host resources. Handles sign-on and authentication of client. Allows reference of devices by name rather than network address.
Transport	Defines addressing and connection protocols for physical devices on network. Guarantees message delivery.
Network	Defines routing and relaying of message packets between networks. Defines protocols for sending status messages to networked computer.
Data Link	Defines protocols to access network for message transmission and reception. Establishes shared use of physical media.
Physical	Defines electrical and mechanical characteristics of physical connection. Defines topology of network.

4.1.1 Ethernet Layer

The Ethernet standard is an integral part of the OSI. Ethernet is a multiple-protocol medium, several different networking protocols can run on it simultaneously. This capability allows the connection of all types of computers without having to install a different type of cable for each connection using a different protocol. Ethernet fulfills the *Physical* and *Data Link* layers of the OSI model.

Orignally designed at Xerox in 1976, Ethernet is a high-speed communications medium that allows information to travel over the network at upto 10 Mbits/sec. Because Ethernet is also a standard adopted by the Institute of Electrical and Electronic Enginneers (IEEE), the majority of computer vendors support the Ethernet standard. A sampling of computers that support the Ethernet standard as standard hardware configuration or with add-on boards are the Cray family, the VAX family from DEC, the IBM PC family, the Sun Microsystems family, the Apollo family, the Symbolics family, Texas Instrument's Explorer I and II and the Masscomp family.

4.1.2 IP: Internet Protocol Layer

The lowest level of communications implemented during this effort was an Internet Protocol in MicroSoft C version 4.0. While this mechanism is intentionally hidden from the user, it is noteworthy in that IP servers are available on virtually all hardware platforms of interest [St88]. Therefore, relying on the existence of an internet protocol does not inhibit the portability of **ORGAL** to heterogeneous environments. The original Internet Protocol code was obtained from the Massachusetts Institute of Technology [Ro86,Sa86]. This version was written in a dialect of C developed at MIT. The first task was to port this code to MicroSoft

C version 4.0. This was necessary because of the three languages currently supported: Gold Hill Lisp, ALS Prolog and PowerLisp. The first two provide a foreign language interface to this version of MicroSoft C. PowerLisp only supports an assembler language interface. In the case of PowerLisp, we also had to develop an assembler language to C interface which allowed us to run our terminate and stay resident C routines in conjunction with that environment. Between the ported internet protocol code and **ORGAL** a small amount of C code had to be written to initialize the network, check the IP message queue for received messages, and to prepare messages for transmission to other platforms. Communication between **ORGAL** and these C routines is accomplished via passing ASCII strings in both directions. Most of the parsing and interpretation of the strings which are exchanged between these routines and **ORGAL** is currently done in Lisp. One of the goals of the future objectives is to push these tasks down onto the C substrate of the interface.

During Phase I only coaxial cable configured with 3Com's Ethernet controller boards on PC/AT workalikes were investigated. The execution of the IP was borne by the PC/AT processor. This combined with the fact that the 3Com controller board acts only as a transceiver limited to 8-bit-wide bus access resulted in a peak 50 Kbits/sec rate for large block transfers. For smaller packets, the effective data transfer rate is on the order of 10 Kbits/sec.

4.2 ORGAL - A Tool for Distributed Processing Development

A complete discription of **ORGAL** is given in Appendix A.

The technical objective of **ORGAL** was to provide developers with an interactive, extensible and intuitive model for distributed application development. Development of such a tool was necessitated by the absence of commercial products which provided these fundamental characteristics. The approach taken was to develop a language based environment for describing distributed computations. This decision was motivated by the observation that compiler technologies could provide hardware independence thus allowing heterogeneous distributed processing. Another design decision was to use an object oriented abstraction for describing a distributed system. This seemed appropreiate as the familiar concept of a network of nodes with applications programs on them mapped naturally into an object oriented paradigm. Simple mechanisms for saving and debugging the state of the system were also design goals of **ORGAL**.

4.2.1 The ORGAL Model of Distributed Computation

ORGAL distinguishes itself from other distributed systems in both the specification and control of a distributed community of processors. Specification is done by defining objects which embody the intended computations and hardware platform characteristics required for a particular distributed computation. Distributed control is facilitated by passing a computational stack along with each message. Execution of a computation is then accomplished by passing a stack from one user defined object to the next. Objects have access to this stack and can dynamically control the computation. A default or cliche behavior for message passing is inherited by all computational objects. This behavior facilitates distributed message passing while providing a mechanism for capturing the result of a foriegn computation asynchronously. While the system does not currently support fully persistent objects, it does maintain a database of object definitions which are accessable by the user.

4.2.2 ORGAL Phase I Results

The original goals of **ORGAL** were to provide a development environment for distributed processing applications. The prototype system implemented for the purpose of proof of concept has accomplished that goal. The prototype has served to clarify many of the issues of distributed processing in general. **Symbiotics Inc.** is currently reimplementing **ORGAL** to facilitate more efficient execution and provide more functionality. **ORGAL** will be released as a commercial product in the fourth quarter of 1988.

A detailed discription of **ORGAL** is included as an appendix to this manuscript. The interested reader is encouraged to read that appendix for a complete discription of the **ORGAL** system.

4.3 KNO: Knowledge Objects

Knowledge Objects, **KNO**, is the substrate component of ACCESS that offers an uniform interface in the form of generic operations on distributed knowledge-based processes. System efficiency and upward evolution is realized by a development environment that allows the application of specialized knowledge representation and processing on the basis of the structure and function of the knowledge.

4.3.1 KNO Design Strategy

Since Symbiotics had to expend a considerable amount of effort in redesigning and recoding the knowledge representation and knowledge processing substrates to be used by ACCESS, we spent some time examining the current state of the art in knowledge engineering tools. In our Phase I effort, we wanted to develop a basic platform that would define an extensiable architecture on which future Phase II effort could continue. We decided to implement from the ground up a knowledge engineering environment with rule-based representation and forward inferencing. This effort would fully explore the issues involved in implementing A.I. technologies in a PC Lisp environment.

The initial design issues of KNO were in response to the lessons learned in enhancing

CommonLoops in the Gold Hill PC/AT environment, the experience one of us (RCP) had just recently gained in porting code from the Lisp Machine to the Sun/Franz Common Lisp environment, and the garnered experience of other people who were involved in the porting wave from Lisp Machines to more conventional platforms that was sweeping "A.I. Alley" in the fall of 1987.

These design decisions were:

1. Garbage collection in Lisp environments, such as Sun/Franz Lisp, VAX Lisp and Gold Hill Lisp, is undependable at best. KNO must achieves almost total self reclaimation of all of its dynamic data structures. This is of strategic importance for any package that is expected to support applications that would be beyond the "toy" level.
2. Do not depend on the native Lisp compiler to make optimizations. Lisp macroexpansion is usually on a "as is" basis. Compiler optimizations such as compile time data type checking, caching of temporary data structures in hardware registers or on the stack instead of of allocating from the heap, and compile time evaluation of code with the inline substitution of constants are not available on conventional machine environments.
3. KNO must have its own compiler for each knowledge representation. Each compiler would generate basic Lisp code that was optimized for the knowledge engine. The resulting code will execute with a minimum of garbage production.

4.3.2 KNO Phase I Results

Foremost, in the design effort of KNO, the replacement for FBI, was to realize the development of a set of abstractions that would allow the application of specialized A.I. technologies on the basis of the structure and the function of the knowledge representated. The goal was to realize an architecture that would allow the functionality of various reasoning technologies typified by Truth Maintenance Systems [Do79,Mc82,Ba83,Dk86], backward and forward chaining inference engines, and various representation technologies typified by semantic nets, Krypton [Br83], rule-based, logic-based and framed-based to be commonly representable as a coherent whole. Efficient integration of a new tool in an existing environment usually requires redesign and modification of the internal data structures and state manipulation machinery of either the tool being incorporated or the incorporating environment or both. During the design of the KNO operation protocols, the subsequent development of the KNO rule-based representation compiler, and the results of developing ORGAL it was realized that a language-based architecture coupled with specialized compilers could be the basis on which KNO would be able to incorporate newly developed or existing artificial intelligence technologies or tools by description instead of the more costly method of reimplementating both large parts of KNO and the package being added.

In the design of the **KNO**, knowledge resources are used as abstractions provided to the developer for the collection of related knowledge under a given context. There are no scheduling schematics associated with the creation of a KNO. Because of the defined set of generic operations that can be performed on a KNO, knowledge processing and scheduling is separated from the knowledge represented in a KNO knowledge base.

A detailed summary and reference manual of a prototype version of **KNO** developed during the Phase I effort is given in Appendix B.

Chapter 5

Summary of Phase I Effort

Traditionally, arguments for developing distributed processing models centered around significant gains in computational efficiency. While this is still a strong motivating force for such work, recent work in artificial intelligence and cognitive science [Mi86,Ru87] suggests that parallel distributed processing is an intricate part of the human problem solving process. In order to take advantage of these observations, the field of distributed artificial intelligence has emerged. The Phase I effort of this proposal was an attempt at prototyping a distributed system of cooperating knowledge sources motivated by the above considerations.

In particular, in order to make advancements in the field of distributed artificial intelligence one must first find a suitable solution to the more general problem of distributed processing. Our first attempt at overcoming this problem resulted in the creation of **ORGAL**. While this has proved to be a powerful state of the art tool for configuring cooperating knowledge sources it still requires that the user have an understanding of distributed processing concepts. As the intended goal is to investigate the various organizational behaviors of problem solvers and not distributed processing issues in general, this requirement seems unsatisfactory. To correct this situation, Symbiotics proposes to build higher level tools which relieve the knowledge engineer from addressing these issues altogether. While the underlying execution of these higher level substrates will make use of **ORGAL** like technologies, this process will be completely hidden from the user.

The naive user should not have to address issues of distributed processing, software developers and sophisticated users need the ability to access the **ORGAL** environment to develop their distributed applications. **ORGAL** is attractive to this community as it will be the first distributed processing tool of its kind available commercially. However, as discribed below, some development must be done to take **ORGAL** from prototype to a commercial product . Most notably, TCP/IP will replace IP and object persistence and fault tolerance must be addressed.

Another finding of the Phase I effort addresses the current state of affairs of commercial knowledge based tools. These tools were developed for stand-alone or centralized processing and are not ideally suited to cooperating knowledge source research. This is largely because the developers of these systems envisioned users embedding their application in the shells themselves. Distributed processing utilizing knowledge sources is better served by functionality which can be embedded in the users application. This observation lead to the development of KNO, an object oriented, open architecture artificial intelligence system, during Phase I.

5.1 Summary of Distributed Communication Results

The choice of Internet Protocol[Cl82] as the low level message passing mechanism was driven by ease of implementation and thus rapid realization of this prototype system. **ORGAL** must support guaranteed message delivery between environments to be of practical use. The commercially released implementation will use a higher level protocol. TCP/IP[Cl82] is the protocol of choice as it guarantees message delivery and is available for all hardware platforms of interest.

5.2 Summary of ORGAL

The feasibility of actually realizing the full power of cooperating knowledge sources is currently limited less by any theoretical problems than by the current state of both available hardware and software platforms. As there are a number of existing knowledge-based and Prolog-based application development tools available which are suitable for various tasks running on various hardware platforms, a truly general system must allow as many of these hardware and software platforms as possible (i.e. a truly heterogeneous distributed processing environment). It is our conclusion that the fundamental tool which determines the feasibility of realizing cooperating knowledge based systems is a methodology for facilitating, interpreting and controlling the information flow among these various hardware and software platforms. The approach we feel is most likely to succeed both technically and commercially is to develop a langauge based environment for doing so.

Our prototype environment, **ACCESS**, solves many of the above problems by supplying the developer with a means, **ORGAL**, of interactively specifying both hardware and software configurations at a single workstation. Modifications as well as new functional descriptions entered at one workstation automatically propagate to the targeted environment.

While **ORGAL** is presently a powerful tool for distributed processing development, there is need for improvement. With the exception of the IP, **ACCESS** is predominantly a Lisp based system. This was purely motivated by the ease with which one can develop language compilers in the Lisp language. It is currently possible to include non-Lisp based packages, as we have done for ALS Prolog. However, agents defined as gateways to these systems must reside on the sending Lisp *Environ* and specialized C code must be written on the target *Environ* to accommodate them.

ORGAL uses asynchronous message passing to allow maximum utilization of processor resources. Currently, there is no mechanism to protect against the possibility of simultaneous access of Agents. This is not a problem in the uni-processing environment of the PC/AT where sequentiality is guaranteed. However, in a multi-processing environment this is a possibility and some mechanism [Mo85,Gr78] must be adopted to guard against it.

While **ORGAL** succeeds in providing asynchronous message passing capabilities there are applications where synchronous message passing and shared memory models are desirable. It is possible to implement these paradigms in **ORGAL** as it is currently designed. However, primitives to support these often cited methodologies could be added to the specification of **ORGAL**, offering a richer functionality to the end user. In addition, similar conclusions can be reached regarding fault tolerance methodologies. It is intended that future enhancements of **ORGAL** will incorporate fault tolerance functionality.

Finally, **ORGAL** currently requires the user to have a certain familiarity with distributed processing concepts. Of all of the suggested models for distributed processing asynchronous message passing models are the hardest to implement, the most general and thus most powerful, and the hardest for programmers, who are more familiar with conventional architectures, to use. Since the domain of this research effort is limited to cooperating knowledge-based systems, we maintain that it is possible to completely hide the burden of distributed processing issues from the application developer. Just as a compiler makes the **ORGAL** language portable and hardware independent while hiding the details of interprocessor communication, another compiler could compile an even higher level declarative language down to the level of an **ORGAL** specification. In this way, knowledge engineers can focus on the artificial intelligence development aspects of cooperating knowledge-based objects without regard for the distributed processing issues which they currently must address in **ACCESS**.

5.3 Summary of KNO

One of the major findings of the Phase I effort addresses the current state of affairs of commercial knowledge-based system development tools. These tools were developed for stand-alone or centralized processing and are not ideally suited to cooperating knowledge-based system development. This is largely because the developers of these systems envisioned users embedding their application in the shells themselves. Distributed processing utilizing knowledge sources is better served by functionality which can be embedded in the users application. This observation lead to the development of **KNO**, an object oriented, open architecture package for knowledge-based system development. Foremost in the design effort of **KNO** was to realize the development of a set of abstractions that would allow the application of specialized A.I. technologies on the basis of the structure and the function of the knowledge representated. The goal was to realize an architecture that would allow the functionality of various reasoning technologies typified by Truth Maintenance Systems, backward and forward chaining inference engines, and various representation technologies typified by semantic nets, Krypton, rule-based, logic-based and framed-based to be commonly representable as a coherent whole. During the design of the **KNO** operation protocols, the subsequent development of the **KNO** rule-based representation compiler, and the development of **ORGAL** it was realized that a language-based architecture coupled with specialized compilers could be the basis on which **KNO** would be able to incorporate newly developed or existing artificial intelligence technologies or tools by description. A language based architecture that can capture the syntactic and semantic description of an artificial intelligence system's supporting substrates will extend the current generation of AI-systems' ability to understand and modify themselves. The same semantic power of expression that allows an application to reason with its own representation presents a homogeneous development environment for the human developer of the application. The homogeneity and semantic descriptive power of a development environment which is language based allows newly developed or existing artificial technologies to be easily incorporated instead of being reimplemented.

Chapter 6

Future Work

6.1 Future Direction In Communcation Substrates

For a long time, fiber-optic cable has been suggested as a replacement for coaxial cable. Fiber-optic cable has a greater range than coaxial, upto 100 kilometers. It is effectively immune to electromagnetic and radio-frequency interference (EMI/RFI), thus reducing noise-induced network errors. This last quality makes installation of fiber-optic cable appropriate in EMI/RFI susceptible areas. Fiber-optic cable can not be short-circuited, making it a viable media in wet environments. A fiber-optic based network is more difficult to tap into illegally, making it more secure than coaxial based networks. Physically, fiber-optic cable is smaller, lighter, more flexible, and more elastic than coaxial cable. These properties simplify cable handling and installation. While the cable itself may cost more, it may be less expensive to install and maintain than conventional coaxial cable.

Symbiotics expects that there are many real-word distributed applications whose peak and perhaps average bandwidth requirements can exceed the 10 Mbit/sec capacity of Ethernet. The Fiber Distributed Data Interface (FDDI) standard for fiber-optic data transmission is in the final stages of specificiation. This standard specifies a bandwidth of 100 Mbit/sec and is a complementary replacement to Ethernet replacing the *Physical* and *Data Link* layer of the OSI model. This forefront technology is currently experiencing rapid evolution. Advanced Micro Devices just recently announced a chip set which represents the first silicon implementation of the FDDI standard [ED87]. Symbiotics expects that network controller boards incorporating the AMD chip set or an equilvalent chip set will become available in 1988. With the network protocol implemented in silicon, access times to distributed memory can approach those of access times of local memory on a PC/AT bus. Depending on FDDI based product availability during the course of a possible future Phase II effort, the feasibility of a distributed communication substrates that use FDDI media will could be developed. The current design of ACCESS's communication substrate the is spread over the *Session* and *Presentation* layers of the OSI and should require minimal changes where it interfaces to the TCP substrate at the *Transport* layer and the IP substrate at the *Network* layer. The largest effort of work would be involved in the restructuring of the IP substrate at the *Data Link* layer of the FDDI where it previously interfaced to the Ethernet.

6.2 Future Directions in Distributed Processing

Based on our Phase I work and current ongoing work **Symbiotics** has identified three separate functional substrates that **ORGAL** will be separated into:

1. **DDSS**
Distributed Data Structure Substrate, **DDSS**, is an extensible substrate designed for the movement of data between heterogenous computing environments. The substrate supplies the underlying communication facilities for distributed data state maintenance, resource access and interprocess communication.

2. **DRDL**
Distributed Resource Description Language, **DRDL** is a representation language for describing resources, maintaining those descriptions, and performing efficient computations with the descriptions in the context in which they are to be used.

3. **ORGAL**
ORGAL is a language for the representation of the interactions and relations of members in an organization. It will be designed to evolve into a language capable of describing organizational representations such as social rules and ordering laws.

6.2.1 Future Directions of DDDS

The **DDSS** will support various distributed data types for communication between multiple distributed processes in computing resource environments that are reliably connected. Data transmission is guaranteed over physically reliable media. Exception handling due to data transmission failures will not be supported by the **DDSS**. Fault tolerance due to layers at and below the *Presentation Layer* of the OSI model (see Table 4.1) is application dependent and is handled at the *Application Layer*. The **DDSS** should be considered a service whose interface is at the boundary between *Presentation Layer* and the *Application Layer* of the OSI model.

The **DDSS** is designed to support communication among multiple processes executing on multiple computational resources. The design will be based on an object-oriented architecture [Co86]. The **DDSS** provides the ability to create multiple independent communication paths and thus will be able to support concurrent processing. The **DDSS** will provide the application developer with an uniform interface based on generic functionality. In order to specify the attributes of the communication, the application requests from **ACCESS** the desired flavor of resource. In this manner, the uniform interface to the **DDSS** shields the developer from the environment dependent implementation details of a particular type of data structure and the implementation of its state accessors and mutators.

Data structure and any future data structure types whose description is captured by the **DDSS** will include descriptions of the implementation of methods which access the state of the data structures. The result will be **DDSS** facilities that can be compiled into their specified implementations as well as a formal description of their semantics. The emphasis of the first part of the **DDSS** development effort will be to investigate and develop the basic features that the **DDSS** requires for supporting efficient distributed artificial intelligence applications and distributed processing applications in general.

In the process of compiling substrates represented in the **DDSS**, assumptions about the implementation representation of the distributed data structures and context and use of the functions that operate on these data structures can be used for optimization. The main design issues that will be addressed are:

1. *What high level abstractions does the* DDSS *need to support implementation descriptions?*
2. *Can a general set of abstractions be found or a general protocol be described that covers a broad range of communication media and methods?*

All accessible nodes must be described in the **DDSS** database. Each node description contains descriptions of all media it can access. Communication between one node and another can only occur when both nodes share a common network. Each network provides a separate address space in which that network's resources can be uniquely referenced. Consequently, all resources on a network can be addressed uniformly. Automatic routing across multiple networks is not provided.

DDSTypes are the distributed data structures which are designed to give the *look and feel* that an application developer might normally experience if working with standard data structures. **DDSTypes** are designed for overall integration with the object oriented style that distinguishes all of the interfaces into ACCESS's functionality. **DDSTypes** present to the application developer a standard model for reading and writing the state of data structures while at the same time minimizing the details of implementating distributed state across multiple computational resources.

The **DDSS** supports an extensible family of **DDSTypes**. Each type of **DDSType** is accessed through a generic set of operations or protocol. The **DDSType** is chosen based on the immediate communication need. In general, **DDSTypes** are not reusuable but are explicitly deallocated after the communication is finished.

Different **DDSTypes** and protocols will be designed for different types of distributed communication needs. From the application developer's perspective, all **DDSTypes** of a certain type obey the same functional contract as specified by their protocol independent of the network they are instantiated on. The actual implementation of a **DDSType** and the implementation of its protocol may differ over different media but the interface will remain the same.

6.2.2 Future Directions of DRDL

DRDL is a structure based representation language for describing structured objects, in this case resources, and their relationships with other distributed resources in the domain of distributed processing applications. The **DRDL** is a knowledge representation language which is a conceptual hybrid of frame-based languages, predicate logic, and functional languages. The eventual goal in developing the **DRDL** is a representation language suitable for capturing high-level descriptions in declarative form that can be used by other tools, services and user applications. The major technical objectives satisified by the **DRDL** are:

Functionality
The level of representation or granularity of a resource description stops approximately at the level of describing *what* a resource does. It is assumed that there will be very little or no representation of *how* a resource does what it does. This suggests that a resource is described by its functional behavior and that a representation of *how* this functionality is accomplished and the definition of the functionality is at best implicit.

Representation of Properties
Although most of the description of a resource is a description of the functionality, there will exist other types of properties that are not classifiable as functional but rather are relational in nature. The description of a resource must be able to uniformly represent both functional properties and properties that are specific to the resource itself.

Classification
Resources naturally break themselves up into classes based on their functionality. These functional classes can be further subdivided by other properties. These properties are usually more specific functionality such as speed, size, manufacturer quirks or any other property deemed important. The description of a resource must be able to represent where the resource resides in the taxonomy of all resources. New resource classes can be defined in terms of aggregates of other resource classes.

Relationships
A resource is not necessarily an autonomous object. A resource can have one or more relationships with another resource. In many cases, the relationship with another resource is an interdependency that will effect how a resource is to be used and thus must be represented. An obvious example of an interdependency is a resource which is the shared memory of two processor modules and resources which are the processors themselves. One processor resource is effected and thus interdependent on what occurs in shared memory due to the other processor. The description of a resource must be able to represent interrelationships in an uniform and efficient manner.

Extensibility
The description of a resource can change over time. For example, a resource can move, cease to exist, come into existence or even have its functional behavior change due to hardware modification. Also of major consideration, is that the description of functionality and classification of resources can evolve. New technology may result in the need to create a new classification. The representation scheme chosen must be able to adapt to changes in the type of description and in the level of description. All of these considerations require that the description be easily accessible by the application developer, by product support and by executing processes.

Introspection

The description of a resource should support the ability of a resource to access its own description. The ability of a resource or any process to access the state of a resource allows upward extensibility of services without extensive coding efforts.

Self-Descriptive

The **DRDL** should be almost completely self-descriptive, relying on a small set of implicitly defined kernel attribute constants and prototype frames to implement the descriptions of all other elements in the **DRDL** universe. Self-description is necessary for any representation that attempts to be declarative. Self-description allows extension and modification of the DRDL's initial base and capabilities.

Type Checking

The **DRDL** will include a set of attribute constants which can be used to describe the type of a class of object sentences. In the case of creating a description it is possible to specify values which are of an illegal data type for the context in which they will be used. Additionally, although the data type of a value assigned to the property of a resource description may be correct, it may be of a magnitude that is physically out of range. Both of these errors, if not checked at the time of the resource description creation or at the time of resource declaration, can go unnoticed until the resource is allocated and used. Obviously, it is better for errors to be caught at the time and place they are created. This is especially true in the case of the distributed application domain where errors during operation can be very costly. The distributed application domain tends to deal with applications which can not tolerate fatal errors during operation. Debugging a propagated error in distributed process is unusually difficult as flow of processing can be indeterminate and global state is not localized.

Separation of Class and Instance

A strict distinction between classes and instances of classes is enforced. The set of all resource classes corresponds to a dictionary of types of resources. A term in the dictionary is convenient for describing a resource as it carries within its definition a set of default properties. The set of all declared resources corresponds to a database of the current configuration of the resources in the environment. The separation of resource class descriptions and instances of classes allows management operations to be performed on one without effecting the other.

Archiving to File

As a system evolves, reorganization of the declared resources and to a smaller extent reorganization of the descriptive classes of the resources is inevitable. The application developer will require archiving tools that allow him or her to incrementally save or load either a configuration database of declared resources or a dictionary of resource class descriptions.

Describing Necessary Conditions

When referencing an unique resource, the description of a resource must contain a set of properties that allow the resource to be distinguished uniquely from all other resources.

The ability to reason about necessary conditions is the ability to find the unique resource that best satisfies a set of given constraints. The ability of a service, such as a resource manager, to be able to reason about necessary conditions requires that the declaration of a resource result in an unique description.

Describing Sufficient Conditions
The ability to reason about sufficient conditions is the ability to find the set of all resources that satisfy a set of constraints from all declared resources in a timely manner. This ability requires that the declaration of a resource support updating the membership lists of classes with which it shares common properties.

6.2.3 Future Directions with ORGAL

The original goals of the **ORGAL** system were to provide users with a high level tool for development of distributed processing applications. Our expectations for any software tool were to be fulfilled for **ORGAL** as well as **ACCESS** in general. For any system these are:

1. Interactive.
2. Intuitive.
3. Extensible.
4. Robust.

Of these we believe we accomplished the first three to a large degree and knowingly ignored the last. Robustness in a distributed system entails addressing issues which do not exist in a single processor environment. These issues fall into two broad categories, persistence of the objects which the user defines and tolerance of the message passing algorithms to network and node failures. While these features were not implemented during the course of Phase I, **ORGAL** was designed with the expectation that these elements of the system would be incorporated in future versions. **Symbiotics** is currently developing a state of the art model consistent with the current design of **ORGAL** which will provide this functional robustness as well as aid significantly in the interactive and extensiable aspects of distributed system development and debugging. These improvements to the language and its compiler are expected to be finished in the fourth quarter of 1988.

In addition to purely technical issues, the predominance of non-Lisp languages in the current software market as well as a percieved desire of many users to access a large number of other commercial software packages, **Symbiotics** is currently porting **ORGAL** to a C based environment. The C language is suitable for manipulating hardware and software at the very lowest levels allowing advanced users versed in C to develop highly application specific software on top of **ORGAL**. For similar reasons **Symbiotics** is actively adding other Lisp environments and hardware platforms to the **ORGAL** domain.

6.3 Future Directions with KNO

KNO will consist of two separate but integrated interface layers that will be presented to the applications developer. The first interface, referred to as **BasicKNO** will present the knowledge representation and knowledge processing functionality usually associated with high-end commercial knowledge engineering tools [KCS86,CG87,In85,Inf87]. The second interface, the KNO Representation Language (**KNORL**), will consist of a language that is semantically rich enough that it allows **BasicKNO** capabilities to be rapidly expanded by the integration of established or experimental AI facilities by describing them rather than reimplementing them.

6.3.1 BasicKNO

BasicKNO differentiates itself from other high-end commercial knowledge engineering tools, such as ART [Inf87], KEE [In85] and KnowledgeCraft [CG87] as it is *designed to be embedded in applications as opposed to applications being embedded in the knowledge engineering tool.* Direct support of input-output functionality by these tools is prohibitively narrow. I/O functionality consists almost entirely of sophisicated user interface utilities that require the AI development tool to be used in an interactive mode. In fact, these user interfaces can account for more than half of the product's development and maintainance costs. In order to perform basic I/O with files, databases and other processes, the application developer must patch in their own application specific I/O functions.

BasicKNO will consist of standard and well-understand knowledge representation and knowledge processing technologies. A Knowledge Object is a knowledge base which is treated as an abstract data type. The KNO interacts with an user or system application only through a small set of operations. This design strategy is based on the functional approach proposed by Levesque and Brachman [Ba83]. This approach requires a knowledge base to be completely specified functionally, ignoring how the knowledge base is implemented.

6.3.2 KNO Representation Language

The other design and development focus of **KNO** will be to construct a **KNO** Representation Language (**KNORL**) for the representation of knowledge-based resources and the operations that can be performed on them. **KNORL** will be a frame-based language in design [St79,Fi85]. The **KNORL** represents the evolution of work currently underway at Symbiotics on knowledge-based object description languages, DAIRLL [Sy87a], adapted to the problem of describing knowledge-based resources. The eventual goal in development of **KNORL** is a language suitable for capturing high-level descriptions in declarative form that can be used by other tools, services and user applications. Like the DAIRLL, **KNORL** will be almost completely self-descriptive, relying on a small set of kernel attribute constants and prototype frames to implement the descriptions of all other elements in its universe of discourse. Self-description allows extension and modification of both **KNORL** and KNO's initial base capabilities by the applications developer. The **KNORL** will serve as the kernel language substrate on which a family of computer aided software engineering tools for distributed artificial intelligence applications will be built.

The representation problem addressed by the **KNORL** has been investigated widely in artificial intelligence research. The main motivating goal in research in Meta-Level reasoning, such as MRS [Ru85], was to solve the problem of combining disparate artificial intelligence paradigms into a coherent architecture. Our design approach is closest to that used in developing Krypton [Br83] and Joshua [Ro87]. We have adopted the functional interface design proposed in Krypton. Krypton proposes the integration of multiple artificial intelligence paradigms through the use of a logic representation and a theorem-prover. Our approach differs from Krypton in that **KNORL** uses a frame-based representation for describing multiple artificial intelligence paradigms. Also, efficient implementation of knowledge processing is not sacrificed as it would be in the Krypton approach. Krypton's theorem proving approach introduces another level of knowledge processing, as it must deduce how to perform any requested deduction. **KNORL** descriptions are transformed at compile time, not at execution time, into knowledge base operation implementations.

The use of frames as a knowledge representation has a solid foundation in fundamental artificial intelligence research. Our use of the **KNORL** has many similarities to the development of frame-based representation language languages. The most famous representation language language, RLL-1, developed by Greiner [Gr80], and later extended by Lenat [Le82], was a frame-based language. RLL-1 was designed to evolve through self-modification (or the help of a programmer) into a language with the desired set of features optimal for the domain on which it was being applied. The original premise was that the language be self-descriptive. Both its representation and the description of its representation was available allowing self-modification. RLL-1 was the core of Lenat's famous AM [Le82] and Eurisko [Le83] learning systems. These systems were remarkable for their limited success in demonstrating learning in the domains of number theory and VLSI design. Haase reimplemented RLL-1, with his language ARLO and later ARLO' [Hs86a,Hs86b]. ARLO contains several novel ideas and software engineering improvements over its predecessor RLL-1. Some of these include a value dependency mechanism that tracks and reconciles changes in properties and their description and a type system which is used to specify restrictions on the dependencies and values of a property defined in ARLO. Haase used ARLO to successfully implement the learning system Cyrano-O [Hs86c] which reproduced most of the results obtained by AM in the domain of number theory.

Our development of the **KNORL** in a future effort will apply the power expressed in a representation language language in a different direction than learning systems. Our approach is to denote the semantics of the **KNORL** on a formal basis using attribute grammar theory [Re82]. Symbiotics is currently developing a representation language for describing distributed resources [Sy87a]. The experience gained from the development of the DAIRL will provide a strong in-house base of experience and techniques that will be of major benefit in the development of the **KNORL**.

In the design of the **KNORL**, precedence will be given to features that support a development environment which allows the incorporation of a wide range of representation and reasoning technologies in a general framework. The primary design goal is an artificial intelligence tool that is semantically rich enough that it allows a development environment to be rapidly expanded by the integration of established or experimental AI facilities by describing them rather than reimplementing them. Additionally, the **KNORL** will realize a full integration of ORGAL and **BasicKNO**. The **KNORL** will allow the capture of descriptions for use by distributed knowledge-based applications in accessing ORGAL's and **BasicKNO**'s functionality. This design will support our major technical objective which is to allow the AI researcher or knowledge engineer to work at the level of structuring the general problem solving activity of the domain application instead of solving programming problems in low-level distributed communication.

6.4 Summary of Future Work

The primary focus of Phase I was to prototype a development environment, **ACCESS**, for Communicating and Cooperating Expert Systems. More generally, this work explored the question of what capabilities are needed in a development environment for embedding distributed knowledge-based systems applications on PC or workstation class platforms.

The result of the Phase I effort was that **Symbiotics** prototyped a general distributed processing environment for a heterogeneous collection of hardware and software platforms. It is these findings which lay the foundation for our future work. In particular, the fundamental question that was resolved was, *What minimal set of generic functionality must be provided by a development system to be able to represent an organization of problem solving agents?*. The Phase I effort has resulted in **Symbiotics** identifying major issues in distributed processing as well as making a significant contribution to resolving these issues.

The next generation of artificial intelligence development environments must be able to incorporate a wide range of representation and reasoning technologies in a general framework. These development environments will enable an intelligent system to reflectively reason about the properties of its knowledge representation, methods of reasoning, and methods of communication. A language based architecture that can capture the syntactic and semantic description of an artificial intelligence system's supporting substrates will extend the current generation of AI-systems' ability to understand and modify themselves. Secondly, and of more immediate need, the same semantic power of expression that allows an application to reason with its own representation presents a homogeneous development environment for the human developer of the application. The homogeneity and semantic descriptive power of a development environment which is language based allows newly developed or existing artificial technologies to be easily incorporated instead of being reimplemented.

The significant innovations of the proposed work relative to other existing research efforts fall into two broad categories; The use of compiler technology to construct an efficient truly hardware independent distributed environment. The creation of a referentially transparent tool, with respect to distributed processing issues, for cooperating knowledge sources.

The components of **ACCESS** supply the framework of an extensible language-based environment for developing distributed artificial intelligence applications. The KNO Representation Language, **KNORL**, provides the language-based capibility for describing knowledge-based objects. The Organizational Language, **ORGAL**, provides the basic core for the implementation of a variety of distributed communication paradigms. The description of the **ORGAL** and **KNO** interfaces is explicit and accessible. Capturing the description of the **DDSS** interface by **KNORL** will serve as a complementary test of the descriptive power of this language. The potential benefits of having the representation of the communication mechanisms of a distributed AI development environment accessible to its reasoning machinery should not be ignored.

The realization of the **ACCESS** environment will result in the ability of its users to robustly develop and maintain large grain distributed processing systems. Although the targeted market is embedded artificial intelligence applications, the ability of the application developer to add their own extensions and access to distributed processing primitives makes the system useful for general distributed processing development.

Appendix A

ORGAL - A Distributed Processing Development Environment

A major impediment to developing cooperating knowledge sources is the absence of a high level tool for distributing these organizations of problem solvers. Furthermore, as our model for cooperating knowledge sources involved Sponsors, Receptionists and other types of meta control objects we needed a substrate for their specification and implementation. In an effort to meet this need, we developed a language **ORGAL** whose compiler handles the low level Internet Protocol interface. This feature allows the user to specify and develop distributed systems in Lisp without concern for networking or communications issues.

A.1 The Language - ORGAL

There are three first class objects (see next section) in the **ORGAL** Environment [1] which the user must familiarize themselves with to develop a distributed application:

1. **Environments - Processing Resource Descriptions**
2. **Agents - Application Gateways**
3. **Messages - Communication and Control Objects**

Environments are defined using DefEnv. It adds hardware platforms (e.g. a PC/AT with a HummingBoard) to the global **ORGAL** database. It allows the compiler for the second object type, Agents, to incorporate knowledge about its resident hardware Environment into its generated data structures. DefAgent is used to define these Agents or application gateways and it is this reliance on DefEnv which allows it to be hardware independent. Once all of the hardware platforms have been described using DefEnv, the DefAgent compiler incorporates the necessary functionality for addressing hardware issues (e.g. network protocols, screen I/O, file management, disk access, etc.) without user intervention. This allows the system user to interactively develop distributed applications at a software level without concern for the underlying hardware details. This feature alone has proved to be a significant advance towards rapid development of distributed prototypes in-house.

Messages are the objects which Agents pass to one another in and across their respective Environments. They contain the state and intent of the distributed computation.

[1] An **ORGAL** Environment is defined to include all Environments, Agents, and Messages accessable on a network. It is the set of all of the Environment, Message and Agent objects known to the system. Agents and Environments are viewed as persistent objects while Messages are of a finite lifetime.

A.1.1 First Class Objects

We define a first class object to be an object which can be written to or read from a storage device and passed as an argument. Since Agents, Environments, and Messages [2] were defined as first class objects above, a mechanism must be provided to allow these operations to be performed on them. There is a default behavior with regards to when these operations are performed on each of the three types of objects. There are only two legitimate reasons to override that behavior: for debugging and to preserve a desired initialization state. Overriding the system defaults is done at definition time by use of the keywords *:scope* and *:lifetime*. The matrix of possible value combinations for these keywords is shown in Figure A.1. In all cases, *:scope* specifies the Environment(s) in which the value of *:lifetime* will specify the duration of the objects existence.

A.1.2 :Lifetime

The attribute *:lifetime* defines the extent to which an object will be preserved in an **ORGAL** Environment. The default value for *:lifetime* depends on the type of object being defined. Currently, Agents have a default value of *:image* while Environments have a default value of *:persistent* for the *:lifetime* attribute. Messages are defined to be objects with a temporary lifetime. A record of Messages may be obtained by the user for debugging purposes by using the macro **Record-Message** [3]. The *:lifetime* attribute may take the following values;

persistent
A *:lifetime* value of persistent causes a copy of the object being defined to be added to both the permanent *:class* database(s) [4] specified as part of the *HOST*'s [5] definition as well as the image of that database(s) in memory.

image
A *:lifetime* value of image causes the system to store the object being defined in the memory resident image(s) of the database(s) and not the permanent database(s).

permanent
A *:lifetime* value of permanent causes **ORGAL** to store the object definition only in the permanent database(s) and not the image database(s).

[2]The symbols Agent(s), Environment(s), and Message(s) will be capatialized in this manual when they are being used to refer to these three classes of first class objects in the ORGAL system.

[3]See the section Messages

[4]A "permanent" copy of the Environment and Agent databases are kept in files on the local node's disk.

[5]*HOST* is a special symbol who's value in an Environment is the name of that Environment in the ORGAL system.

Figure A.1: ORGAL Keyword Combination

:lifetime	:scope LOCAL	:scope GLOBAL
PERMANENT	'Host' file only.	Stored in files on all Environments known to 'Host'
PERSISTENT	'Host' memory and file only.	Stored in memory and files on all Environments known to 'Host'.
IMAGE	'Host' memory only.	Stored in memory on all Environments known to 'Host'.

A.1.3 :Scope

The attribute *:scope* defines which Environments an object definition will side-effect. The default value of *:scope* for Agents is local while the default value for Environments is global. A Message's scope is defined to be all of the Environments which the Message traverses in its lifetime. The two possible values for the *:scope* attribute are:

local
A *:scope* value of local restricts the extent of the above *:lifetime* values to the Environment on which the object is created, the current value of *HOST*.

global
A *:scope* value of global extends the effect of the *:lifetime* value to all foriegn Environments known to the current *HOST* in the case of Agents and Environment definitions.

A.1.4 Generic Operations on First Class Objects

Messages, Agents and Environments are all first class objects. We differentiate one from another because their behavior and intent are distinguishable. The following functions differentiate between them by use of the keyword argument *:class*. The following functions are provided to operate on the object databases;

```
(List-Members &key (:scope global)
                   (:lifetime persistent) (:class Environment))
```

Returns a list of all members found in the specified :class database(s) and/or image(s). *:class* can be any first class object; Environment, Agent, or Message. *:scope* and *:lifetime* may have any of the values defined in Figure A.1.

```
(Consistentp &key (:scope global)
                  (:lifetime persistent) (:class Environment))
```

Consistentp is a function with optional arguments which checks all of the specified *:class* databases for consistency. It returns *t* if successful and otherwise returns a list of *:class* objects which are inconsistent with the *:class* database known locally to the current *HOST*. *:scope* and *:lifetime* can take any of the values defined in Figure A.1.

```
(Reset &key (:scope global) (:class Environment))
```

Causes all of the *:class* objects known to *HOST* to reload their permanent copies of the Environment database into the current image. *:scope* can be local or global, *:class* can be either environment or agent.

```
(Destroy &key (:scope global)
              (:class Message) (:lifetime persistent))
```

Delete the database(s) and or image(s) of the object *:class*.

```
(Describe arg &key (:class Agent))
```

Returns the definition of object named arg in the object *:class* specified. Environments may be specified by prefixing the Environment name followed by :: to the value of arg (e.g. FOO::MAIL).

Each of the *:class* objects will now be described in detail followed by an example to demonstrate their interactions.

A.1.5 DefEnv - Adding Hardware Platforms to the ORGAL Database

ORGAL maintains three databases internally. One contains the hardware platforms which are accessible in the users' **ORGAL** Environment. An identical copy of this database resides on all of these hardware platforms by default. The use of Env, an abbreviation for Environment, is intended to denote that this object constitutes a comprehensive hardware environment for supporting distributed software applications.

DefEnv is the top level macro for adding a hardware specification to the database:

```
(DefEnv Name :attribute1 value1 ... :attributeN valueN)
```

Name must be a symbol which uniquely identifies this hardware platform in the global **ORGAL** database. The attribute/value pairs which follow this name define the characteristics of the hardware. The host Environment name is bound to the special symbol *HOST*. The following list of attributes must be supplied:

:type *symbol*
The type value specifies the commercial name of the platform being defined (e.g. PC-AT). This object will inherit all of this objects attribute values not defined at this time. Currently supported types can be found in the special symbol *HARD-TYPES*.

:net-address *string*
Net-address is the network address which this hardware platform is known by on the ***:net-type*** network (e.g. "00.44.50.14").

The following attributes are optional;

:processor *string*

The processor value defines the processor of this hardware platform (e.g. 80286). Currently supported processors can be found in the special variable *HARD-PROCESSOR*.

:memory *number*

The memory value refers to the amount of extended memory available to the above processor. Values should be specified in bytes (e.g. 6784000).

:permanent-memory *string*

Memory (e.g. "c:") which will be used to store permanent copies of the **ORGAL** databases.

:timeout *number*

Default timeout in seconds for an acknowledgement from this device. This value is overridden for a particular Agent transaction if it possesses a *:timeout* attribute value itself (see DefAgent below) The default value for timeouts is bound to the symbol *Default-Timeout* which is unique to each Environment.

:operating-system *symbol*

The operating-system value denotes the operating-system present on this platform (e.g. DOS). Currently supported operating- systems can be found in the special variable *HARD-OS*.

:net-type *symbol*

Net-type defines the type of network which this hardware platform directly connected to (e.g. EtherLink-3Com). Possible values can be found in the special variable *HARD-NET*.

:agent-database-file *pathname*

The pathname of a file in which to store Agent definitions. This database is loaded when an **ORGAL** Environment is initiated or a call to the generic function Reset is made.

:environ-database-file *pathname*

The pathname of a file in which to store Environment object definitions. This database is loaded when an **ORGAL** Environment is initiated or a call to Reset is made.

:message-database-file *pathname*

The pathname of a file in which to store Message definitions. This file can only be accessed by the function Tell and the generic operators List-Members and Destroy.

:scope *symbol*

Scope defaults to global indicating that this object will be distributed to all Environments known to the current value of *HOST*. The other possible value is local which causes this DefEnv to alter only the database local to this *HOST*. This latter option is only provided for debugging and should be used with extreme care as it can easily destroy the integrity of the **ORGAL** system.

:lifetime *symbol*

Lifetime defaults to persistent which indicates that this object will be stored in both the permanent copy of the database(s) and the current image of the database(s). The other two possible values are permanent and image. A value of permanent causes this DefEnv to be only stored in the permanent copy of the Environment database(s). A value of image causes this DefEnv to be stored only in the current memory resident database image(s).

Additional keywords may be added at the user's discretion and will appear when the user requests a platform description (see the generic operator Describe).

A.1.6 DefAgent - Adding Application Gateways to the ORGAL Database

ORGAL also maintains a database of accessible software application gateways resident on the Environments specified by DefEnv. Each of these databases is unique to its locality by default.

DefAgent is the top level macro for putting gateways to a users application software into an **ORGAL** database. The name Agent is intended to denote that this object will act as an Agent in the **ORGAL** distributed Environment for a user supplied application:

```
(DefAgent Env::Name :attribute1 :value1 ... :attributeN :valueN)
```

Name must be a symbol which uniquely identifies the software being described on the hardware platform specified by Env. Env must be the name of a hardware platform defined using DefEnv. If Env:: is omitted, the **ORGAL** compiler will assume that this Agent is to remain resident in the Environment in which it is being defined. The list of attribute/value pairs which follow allow the user to specify the characteristics of the object being defined. There are no required attributes. The following attributes are optional:

:type *symbol*

The type value must be either the symbol 'generic or an Agent name previously defined by the user. The Agent being defined then inherits all attributes from this type not explicitly defined in this DefAgent form through single inheritance.

:gateway-for *symbol*

The gateway-for value should be the symbol 'application or the name of a user specified Agent. If an Agent name is supplied here, it will receive all Messages sent to the Agent being defined by the enclosing DefAgent.

:in-filter *form*

The in-filter value must be a single s-expression. The s-expression will be evaluated in an environment which contains all of the attribute values contained in a message *(see section Messages)*. The evaluation will take place when this agent receives an incoming messge.

:out-filter *form*
The out-filter value must be a single s-expression. The s-expression will be evaluated in an environment which contains all of the attribute values contained in a message *(see section Messages)*. The evaluation will take place when this agent receives an outgoing messge.

:result *form*
The result value must be a single s-expression. The s-expression will be evaluated in an environment which contains all of the attribute values contained in a message *(see section Messages)*. The evaluation only takes place when this agent is an applicaton agent *(see Agent Types)*.

:timeout *value*
The amount of time this agent should continue to try to pass a message to its *:gateway-for* value.

:scope *symbol*
Scope defaults to local indicating that this object will only be known on the current value of *HOST*. The other possible value is global which causes this DefAgent to alter all the databases known to this *HOST*. See the section *:scope*.

:lifetime *symbol*
Lifetime defaults to image which indicates that this object will be stored in the current image(s) of the database(s) of Agents. The other two possible values are permanent and persistent. A value of permanent causes this Agent to be stored only in the permanent copy of the Agent database(s). A value of persistent causes this Agent to be stored in both the memory resident database image(s) and the permanent database(s). See the section *:lifetime*.

In addition to the above, the user may supply their own attributes. These attribute/value pairs will be displayed when the user requests a description of the Agent (see the generic operator Describe).

A.1.7 Messages

In **ORGAL** Messages are first class objects. They are the only arguments which an Agent receives. An **ORGAL** Message is itself an object which contains slot/value pairs. Agent attributes (e.g. :in-filter) values are evaluated in an environment, called a *Message Closure*, which contains the slot values of the Message bound to the symbols below. Furthermore, after the evaluation of an attribute value, the Message slots are rebound to the values of these symbols allowing the user to change the state of the Message object. The accessible slots of a Message are:

target *symbol*
The name of the Agent who will receive this Message next.

diiection *keyword*
Either *:in* or *:out*. When the value is *:in*, this Message is moving towards the application Agent. When the direction is *:out*, this Message has already reached the application Agent and is moving back towards the originating Agent (for a discussion of application, intermediate and originating Agents see the section *Agent Types*).

result -
By default this symbol is bound to what the *:result* of the application Agent returns.

travel-log *list-of-symbols*
While the value of this Message's direction is *:in*, travel-log is a list of all the Agents who have received this Message. When the value of direction is *:out*, travel-log is a list of all the Agents who will receive this Message on the way back up the chain to the originating Agent.

args -
This slot contains the arguments which Tell passed to the originating Agent (see the Section Agent Types).

priority *number*
Messages sent from one Environment to another are automatically put on a priority queue. The default priority given to a Message is bound to the atom *Default-Priority*.

The user may reference these slots by using the symbols listed above. If the user side effects a slot (e.g. (setq travel-log (cddr travel-log))) the new value will become the Message slot's value.

Messages may be written to a database for the purposes of debugging (see the generic functions Destroy and List-Members). It is meaningless to load a Message database into an image. The function Record-Message should be used to store a Message;

```
(Record-Message ...)
```

Record-Message is a macro which the user may wrap around any attribute's value. It will cause a Message to be time-stamped and written to the file specified by the Environment's *:message-database-file* attribute on the Environment in which it is evaluated.

A.1.8 Tell - Creating and Passing Messages

There is only one way for the user to create a Message object and pass it to an Agent for processing. This is done by application of the function Tell to the target Agent and the desired arguments for that Agent.

```
(Tell Env::Agent-Name arg)
```

Env must be an Environment previously defined by a DefEnv. If the Env:: is omitted, that is just an Agent-Name is supplied, Tell will assume that the Agent resides in the Environment in which this Tell form is executed, the current value of *HOST*. Agent-Name must be a symbol which is the name of an Agent previously created with DefAgent. Arg can be anything which can be coerced into a string.

The algorithmic behavior of Tell is as follows:

1. Find the Agent referred to and instantiate it if an instance of it does not already exist.
2. Instantiate a Message object with the target slot bound to the Agent name and the args slot bound to a list of the arguments specified in the Tell form.
3. Pass the Message object to the *:in-filter* of the specified Agent.
4. Return the Message object instance.

Step 1 of this algorithm brings to light another design feature of **ORGAL**. The **ORGAL** language is demand driven. It delays evaluation to avoid making superfluous instantiations.

The database of Agents actually contains the uninstantiated native code generated by the **ORGAL** compiler. It is this feature which allows support of inheritance across Environments.

A.1.9 Agent Types - Runtime Message Handling Behavior

Agents are objects used to define gateways to user applications. These Agent objects are stored in databases local to their specified Environments by default. The behavior of an Agent object is not realized until it is run or fired.

There is only one way to cause an Agent to fire. It must receive a Message. There are two mechanisms by which an Agent can receive a Message. Execution of a Tell form explicitly naming an Agent causes a Message object to be generated and passed to that Agent. Agents invoked in this manner are called **Originating Agents**, unless their *:gateway-for* value is the symbol 'application (see below), because they are the first to see a Message. Alternatively, if an Agent's name is the value of the *:gateway-for* attribute of another Agent, it will receive all Messages that this other Agent receives. Recall that the *:gateway-for* attribute of an Agent may be either the symbol 'application or the name of another Agent. If the *:gateway-for* value is the symbol 'application, then such an Agent is called an **Application**

Agent. If the *:gateway-for* value is the name of another Agent, and the Agent with this *:gateway-for* value is not an **Originating Agent** (i.e. was not invoked by Tell), then this Agent is referred to as an **Intermediate Agent**. The behavior of these three types of Agents is different with regards to Message passing. We will now examine each of these behaviors.

A.1.10 Originating Agents

Any Agent which is not an **Application Agent** can be an **Originating Agent**. Such Agents are distinguished both by being explicitly named inside of a Tell form and by not having the symbol 'application as their *:gateway-for* attribute value. Their algorithm with respect to Message passing is as follows:

1. Immediately return nil to the callers Environment. If the Message received has a direction slot value of *:in* go to step 2. If the direction slot value is *:out*, go to 3.
2. Evaluate this Agents *:in-filter* value. Go to 4.
3. Evaluate this Agents *:out-filter* value. Go to 5.
4. Pass the Message to the value of this Agents *:gateway-for* attribute.
5. Return nil, we are done.

A.1.11 Intermediate Agents

Only Agents which receive Messages from other Agents and have a *:gateway-for* attribute value which is the name of another Agent can be considered **Intermediate Agents**. Their Message passing algorithm is as follows:

1. Immediately return nil to the callers Environment. If the message received has a direction slot value of *:in* go to step 2. If the direction slot value is *:out*, go to 3.
2. Evaluate this Agents *:in-filter* value. Go to 4.
3. Evaluate this Agents *:out-filter* value. Go to 5.
4. Pass the message to the value of this Agents *:gateway-for* attribute.
5. Pop the value of the received message's travel-log slot and pass the message to that Agent.

A.1.12 Application Agents

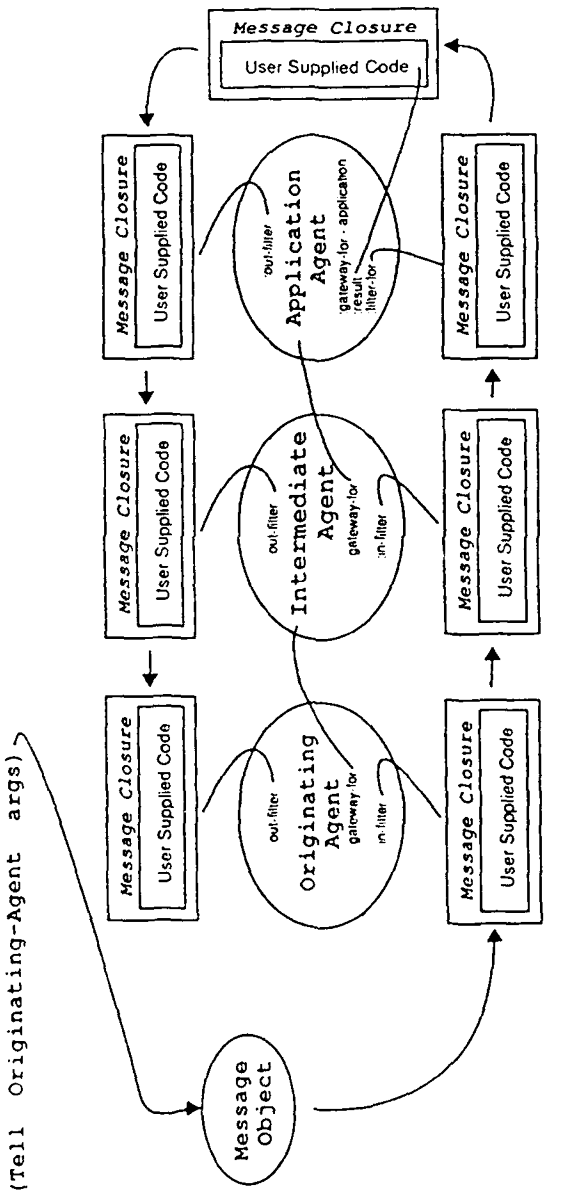
An **Application Agent** is any Agent whose *:gateway-for* value is the symbol 'application. These are the only Agents which evaluate their *:result* attribute by default. The algorithm for their message passing behavior is given below:

1. Evaluate this Agents *:in-filter* value.
2. Evaluate this Agents *:result* value.
3. Set the Message slot value to *:out* and bind what is returned by step 3 to the Message's result slot.
4. Evaluate this Agents *:out-filter* value.
5. If the value of the Messages travel-log slot is nil, then we are done. If the travel-log value is a non-empty list, pop the travel-log list and pass the Message to that Agent.

Figure A.2 graphically depicts the behavior of these three types of Agents. The important consequences of this facet of the **ORGAL** Environment are:

1. The locality of these Agents has no affect their Message passing behavior. These three Agents could reside on one, two or three different processors and the mechanics of their behavior will not change. This allows the user to develop and debug individual Agents on a single processing Environment without making use of the network.
2. The nature of the user supplied code, provided it does not explicitly side-affect the Message object, is also irrelevant to their Message passing behavior. The user supplied code could call an expert system, a prolog system, a fast fourier transform, accounting routines, etc.
3. Step 1 of all of the above algorithmic behaviors always begins by returning a value of nil. This has two significant consequences:
   (a) Control of the processing Environment is always immediately returned to the caller when crossing Environments. This allows that Environment to continue or initiate other tasks. This kind of non-deterministic asynchronous Message passing is extremely powerful [Ag87].
   (b) Results of any work done for value, as opposed to effect, by any **Intermediate** or **Application Agents** must be preserved in the Message object itself. Eventually, the *:out-filter* of the **Originating Agent** will receive the final Message object. It is this *:out-filter*'s responsibility to perform any desired tasks to be done for effect in its Environment. Note that the ultimate return of Messages to the *:out-filters* of Agents gives the **ORGAL** system an asynchronous re-entrant flavor. This aspect of **ORGAL** is unique among distributed processing systems.

Figure A.2: ORGAL Agent-Agent Relationships



The function Tell results in a Message Object being instantiated. The lines with arrowheads indicate the path of the Message Object from Agent to Agent. Lines without arrowheads depict pointers from attributes to their values.

A.1.13 A Simple Example

Agents are a convenient way of defining a sequence of events intended to take place on several processors or Environments. They provide a high level mechanism for running software available on hardware accessible on your network. In order to accomplish this task, they pass Messages which contain the state of the transactions they intend to perform. The flow of Messages from Agent to Agent results in a sequence of events. While the user defines the functional results of these events, the mechanics of Message passing are invariant to the application or hardware involved. An example best demonstrates this feature of **ORGAL**.

Lets assume that there are three hardware platforms in the global **ORGAL** database: A, B, and C. These platforms were added to the database by DefEnv. Furthermore, on each of the above platforms, there resides a set of Agents. These Agents were defined in the following manner:

```
 (DefAgent A::Delegated-Member-Test
  :documentation "Split a list in half and check for
                          membership of an item in each half on
                          different processors."
  :gateway-for B::Member?
  :in-filter (let*
    ((item       (car args))
    (seq         (cdr args))
    (list-length (length seq))
    (half-length (truncate list-length 2))
    (first-half  (subseq seq 0 half-length))
    (second-half (subseq seq (+1 half-length))))
;pass the first half to C::Member?
       (tell A::Member-Test-2 item first-half)
;change the Message slot args to be the second half for B::Member?
       (setq args (list item second-half)))
  :out-filter (if result
  (progn
    (format t "~% ~s is a member." (car args))
;store the results of B::Member
                              (push results *member-test-results*))))

 (DefAgent A::Member-Test-2
    :type A::Delegated-Member-Test
    :gateway-for C::Member?
    :in-filter nil)

 (DefAgent B::Member?
    :documentation ''An applications gateway for the member function.''
    :gateway-for application
```

```
   :result (let ((item (car args))
                 (list (cdr args)))
               (member item list)))

 (DefAgent A::Member?
   :type B::Member?)
```

Evaluation of these forms causes these agents to be added to the agent databases associated with their *environ*.

If the user now executes the form:

```
(tell A::Delegated-Member-Test 10 '(1 2 3 4 5 6 7 8 9 10))
```

A message with the following slot values will be created:

```
taget          - 'A::Delegated-Member-Test
direction      - :in
result         - nil
travel-log     - nil
args           - '(10 (1 2 3 4 5 6 7 8 9 10))
priority       - *Default-Priority*
```

The *:in-filter* for Agent A::Delegated-Member-Test will receive this Message and its value will be evaluated in an environment with the above symbols bound to the listed values (i.e. a Message closure). As a result of this evaluation, the list '(1 2 3 4 5 6 7 8 9 10) will be split into two lists, one of which is explicitly passed to A::Member-Test-2. This Agent was defined with a *:type* attribute value of A::Delegated-Member-Test. Recall that an Agent inherits all of its attributes from the Agent specified in its *:type* attribute with the exception of those explicitly defined in its DefAgent form. The *:in-filter* attribute of A::Member-Test-2 is explicitly set to nil here so that the list bound to the Message args slot will not again be split into two. The *:out-filter* however, is inherited from A::Delegated-Member-Test as it correctly stores the result in the same special variable.

A::Member-Test-2 is an originating Agent by definition. The *:in- filter* of A::Member-Test-2 is nil so nothing is done to or with the Message. The Message is simply passed on to it's *:gateway-for* value, C::Member?. Recall that all Agents immediately return control to the sending Environment. Once the Message is passed from Environment A to Environment C, control is returned to A.

A::Delegated-Member-Test is now free to continue. It resets the args slot of the current Message to the first half of the list. As the *:in-filter* has completed its evaluation, the

Message is now passed to its *:gateway-for* value B::Member?. In a manner identical to C::Member?, control is now immediately returned to Environment A.

Two originating Agents, Delegated-Member-Test and Member-Test-2, in Environment A have each passed a Message to different application- Agents on different Environments. Eventually, these foreign application Agents will complete their *:in-filter*, *:result*, and *:out-filter* sequence as prescribed by the algorithm defined above. Having done so, they will send the resulting Message object back to the *:out-filter* of their originating Agents. Note that there is no ordering of events imposed here. Which originating Agent receives the resulting Message object first is non-deterministic. If the user application requires that both halves of the list have been checked for membership before proceeding, it is up to the user to wait (e.g. loop) until the list bound to *member-test-results* is of length two.

Eventually, A::Member-Test-2's *:out-filter* will receive a Message with the result slot bound to t while A::Delegated-Member-Test's *:out-filter* will receive a Message with the result slot bound to nil. The value of *member-test-result* will either be the list (t nil) or the list (nil t) depending on who finishes first.

A.1.14 A Methodology for Development

Every programming environment suggests its own style for development. New **ORGAL** users are strongly urged to read and utilize the following procedures while developing distributed applications.

1. The **ORGAL** model is based on the assumption that the Environment databases are consistent and global. While the system allows you to break the model through the use of the *:scope* and *:lifetime* keywords this is not the intended use of these features. In general, the user is strongly discouraged from using a *:scope* value of *:local* with DefEnv. The default values for *:scope* and *:lifetime* are usually (see below) reasonable for normal operating conditions.

2. Always set the *:lifetime* option of all objects to a value of *:image* until you are satisfied that the system is debugged and behaving as desired. At this point, it is safe to save your definitions permanently by specifying a *:lifetime* value of *:permanent*. Remember it is an error to try to redefine an existing object. You must first delete it by using UnDefAgent or UnDefEnv.

3. **ORGAL** does not currently provide the user with an editor. A future release will contain a structure editor for DefAgent and DefEnv. Until that feature becomes available we suggest that you do your development in an Emacs style editor making use of the Form Evaluation feature (usually Control-X Control-E). True beginners may find it helpful to start by modifying existing definitions in the FREE-AG.LSP file to get a feel for the systems behavior.

4. Unless you are debugging your system, it is meaningless to keep copies of Message objects in the Message database(s). Remember that you must explicitly delete these databases with the generic operator Destroy with a *:class* value of message.

5. **ORGAL** is not currently fully fault tolerant. Should a node become non-functional for any reason during the course of a computation it is up to the application code to handle the exception.

6. Error handling is not explicitly provided as part of the **ORGAL** Environment. This is because of the lack of an error system in the current Common Lisp standard. See the examples in the file FREE-AG.LSP and the documentation on errors of the Lisp you are running for suggestions on addressing this issue.

Appendix B

Basic KNO Reference Manual

B.1 Basic KNO Design Model

KNO, KNowledge Objects is the substrate of **ACCESS** for the representation of and processing of knowledge. The definition of a Knowledge Object, referred also to as a *KNO*, is conceptionally very similar to an Actor[Ag87] or a Deliberate Agent[Ge87]. All KNOs define their external functional behavior with their environment through a small set of operations, **KNO Operations** or **KNOOP**. The major component of the *KNOOP* defines a full and uniform functional access to a KNO's internal state. The internal state of a KNO represents the KNO's conceptualization of its universe and as such is a knowledge base. In its most simplest conception, a KNO can be thought of as a database of knowledge structures. The implementation of the knowledge represented as state in a KNO is hidden by the functional definition of the *KNOOP*. The operation, *TELL*, for example, is used for performing all forms of assertion on a KNO's knowledge base.

In order for the KNO to support a broad spectrum of knowledge representation and knowledge processing the internal architecture must be accessible. The functional architecture of a KNO, or the entire class of KNOs, can be extended by defining a new **KNO Protocol, KNOPROT**. Alternately, the functional defintion of a KNO's behavior can be retained while its internal architecture can be altered by defining a new **KNO Implementation, KNOIMP**. Both a *KNOPROT* and a *KNOIMP* define the behavior of a *KNOOP*. The determination of which *KNOPROT* or *KNOIMP* is used for a given *KNOOP* is specified by the type of the knowledge the KNO is being requested to process.

The external representation of all declarative knowledge presented to a KNO is in *predicate sentence form*. A *KNO predicate sentence* is formed from a n-ary predicate constant, ρ and n terms $\tau_1 ... \tau_n$ in the following form:

$$(\rho\ \tau_1 ... \tau_n).$$

The type of the knowledge is specified by the *predicate* of the *predicate sentence*. A specific *KNO Protocol* and *KNO Implementation* are specified for a specific *predicate* or class of predicates. This method for determining the type of the knowledge being processed is itself a protocol and in the current implementation it is the *default KNOPROT* for all *KNOOP*. All predicates, unless otherwise specified, assume a *default KNOPROT* and a *default KNOIMP* where such is required. For example, the *default KNOPROT* for the KNOOP *TELL*, operating on the Knowledge Object *Agent* for the assertion `(TELL 'AGENT '(ON BLOCK-A TABLE))` is:

1. Determine if the KNO `'AGENT` is known. If `'AGENT` is known, send the message `(TELL`

'AGENT '(ON BLOCK-A TABLE)) to 'AGENT.

2. AGENT checks and verifies that ON is a known predicate.

3. The KNO AGENT finds and dispatchs on the declared *KNOIMP* for the TELL operation for the predicate ON. In this case the *KNOIMP* for ON is the *default KNOIMP* for TELL For the current implementation of Prototype KNO the default KNOIMP for TELL is forward chaining inference.

4. Store ground predicate sentence, '(ON BLOCK-A TABLE) in knowledge base of the KNO named AGENT, if and only if the sentence is not already present.

5. If the sentence is new, send it into the discrimination net for forward resolution.

B.1.1 KNOOP: Basic KNO Operations

The design and architecture of Knowledge Objects is based on an object-oriented methodology. The *KNOOP* defines a generic and highly abstract set of knowledge processing operations. The KNOOP *TELL* is the operation for asserting any knowledge to a KNO. The KNOOP *RETRACT* is the operation for removing knowledge from a KNO. A KNO specifies what type of knowledge processing will be executed by the type of the knowledge. The type of the knowledge is determined by the predicate constant of a predicate sentence.

TELL

TELL stores a predicate sentence in the knowledge base of a KNO using the TELL operation protocol implemented for the predicate. The arguments to the TELL operation are a KNO instance and a predicate sentence. In general TELL performs knowledge processing on the referenced KNO using forward inferencing. TELL may accept optional arguments as required by the definition of the TELL operation for the predicate. The predicate sentence argument of a TELL operation may not contain any free variables.

ASK

ASK retrieves all predicate sentences in the knowledge base of a KNO which can be matched or inferred using the ASK operation protocol implemented for the predicate. The arguments to the ASK operation are a KNO instance and a predicate sentence. ASK can be considered a query to the referenced KNO's database. In general ASK performs knowledge processing using backward inferencing. The predicate sentence argument of an ASK operation may contain free variables. ASK may accept optional arguments as required by the definition of the ASK operation for the predicate.

DELETE

DELETE removes a predicate sentence in the knowlege base of a KNO using the DELETE operation protocol implemented for the predicate. The arguments to the DELETE operation are a KNO instance and a predicate sentence. In general DELETE performs knowledge processing on the referenced KNO using Basic KNO's truth maintenance system. DELETE may accept optional arguments as required by the definition of the TELL operation for the predicate.

RETRACT
RETRACT removes a predicate sentence in the knowlege base of a KNO using the RETRACT operation protocol implemented for the predicate. The arguments to the RETRACT operation are a KNO instance and a predicate sentence. In general RETRACT performs knowledge processing on the referenced KNO using Basic KNO's truth maintenance system. RETRACT may accept optional arguments as required by the definition of the RETRACT operation for the predicate. Unlike DELETE, the default functional contract of RETRACT is to not only remove the referenced predicate sentence but also remove all inferences that depend on the assertion of the predicate sentence.

EXPLAIN
EXPLAIN retrieves the justification of a predicate sentence in the knowlege base of a KNO using the EXPLAIN operation protocol implemented for the predicate. The arguments to the EXPLAIN operation are a KNO instance and a predicate sentence. In general EXPLAIN performs knowledge processing on the referenced KNO using Basic KNO's truth maintenance system. EXPLAIN may accept optional arguments as required by the definition of the TELL operation for this predicate.

B.2 Current Level of KNO Development: Prototype KNO

B.2.1 Known Bugs

The first compilation of a rule results in a memory error in GH/HB. This is due to a structure used by the rule compiler that is referenced before it has been properly defined. In other Lisp environments this error does not occur. To recover from this error, exit from the debugger and attempt to recompile the rule again.

DELETE fails on conjunctive rules. The protocol for reclaiming all data structures is not yet fully operative for rules with more than one predicate sentence clause in the antecedent body. The only recovery from this is not to use DELETE. Using DELETE to remove ground predicate sentences from the knowledge base that are referenced in rules with conjunctive clauses will corrupt the garbage collector and thus the entire KNO development environment.

B.2.2 Creation Functions

Create-KNO *KNOId*
Create-KNO creates an instance of a KNO class.

Define-Predicate *KNOId Predicate-name arity*
Define-Predicate defines a new predicate for a class of KNOs or a particular KNO instance. The predicate must be defined for a KNO class or instance before it can be referenced in a KNO operation such as TELL or ASK. The arguments for Define-Predicate are a KNO instance or KNO class, a predicate symbol and optional initialization keywords.

Define-Rule
Define-Rule defines a new Rule for a class of KNOs or a particular KNO instance. The arguments for Define-Rule are a KNO instance or KNO class, a Rule symbol, the rule body and optional initialization keywords. The complete specification for defining a rule is described in detail in the section *Rule Syntax*.

B.2.3 Destruction Functions

Clear-KB *KNOId*
All predicate sentences are deleted from the knowledge base of *KNO*.

Clear-Rule-Base *KNOId*
All rules are deleted from the rule base of *KNO*.

B.2.4 Description Functions

Describe-Statistics *KNO*
Display the knowledge processing statistics of *KNO*. Currently only rule based forward inferencing statistics are logged. The display produced for
`(DESCRIBE-STATISTICS 'AGENT)`
are:

```
Clause Element Matching
 Successful Element-matches                      : 1000
 Failed Element-matches                          : 1000
 Total Element-matches                           : 2000

Entire Pattern of Clause Matched
 Successful Matches                              : 999
 Failed Matches                                  : 1
 Total Matches                                   : 1000

Clause - Clause Unification
 Successful Merges                               : 0
 Failed Merges                                   : 0
 Total Merges                                    : 0
```

```
Rules Triggered                                   : 999
Rules Fired                                       : 999
```

The statistic *Clause Element Matching* counts the number of constant terms in all predicate sentence patterns of a *KNOId*. All the constant terms of a predicate sentence pattern must be matched before the entire pattern of a clause is considered successfully matched. This event is measured by the statistic *Entire Pattern of Clause Matched.* A rule condition body is composed of one or more clauses (predicate sentences). Multiple clauses in a rule's condition body specify a conjunctive sentence. The statistic *Clause - Clause Unification* counts the number of times a pair of conjunctive clauses are satisfied. When a set of predicate sentences are found that satisify all the clauses of a rule then a rule is satisified. This event is counted by the statistic *Rules Triggered.* When a satisfied rule is scheduled and its antecedent is evaluated the rule is said to have *fired.* This event is counted by the statistic *Rules Fired.*

Describe-Rule-Base *KNOId &key (brief 't)*
Displays all the rules currently defined in the rule base of *KNOId.* An argument of *nil* to the keyword *brief* will result in a verbose description of the rules.

Describe-Predicate-Base *KNOId &key (brief 't)*
Displays all the predicates currently defined for *KNOID.* An argument of *nil* to the keyword *brief* will result in a verbose description of defined predicates.

Describe-KB *KNOId*
Displays all the ground predicate sentences in the knowledge base of *KNOId.*

B.2.5 KNOOP: Knowledge Base Operations, Prototype Version

TELL *KNOId*
Fully implemented as forward inferencing with a rule-based representation.

DELETE *KNOId*
Fully implemented as a dependency net of all inferences.

DESCRIBE *KNOId*
Implemented as the various *Describe-x* functions. *Describe-KB* and *Describe-Statistics* are examples of these functions. These functions will be removed from the interface at a later date and replaced by the generic operation **DESCRIBE**.

RETRACT *KNOId*
Not yet implemented.

ASK *KNOId*

Not yet implemented.

EXPLAIN *KNOId*
Not yet implemented.

B.2.6 Miscellaneous Functions

Reset-Statistics *KNOId*
Sets all knowledge processing statistics counters of *KNOId* to zero.

Gc-Statistics
Display the statistics of the **KNO** environment garbage collector. There is one garbage collector and central data structure reclaimation facility per computation node. GC Statistics records allocation and deallocation events for all data structure types used by a **KNO** environment by all currently active *KNOs*. The number of KNO data structures that can be allocated from each type's reclaimation heap rather than being allocated from the available memory is also recorded. This statistic is called *length*.

B.3 Rule Syntax

The BNF form for specifing syntax has been adopted. In our notation:

term is to denote that *term* is optional.

- {*term*}*is to denote that *term* is optional and may appear more that once.
- {*term*}$^+$ is to denote that *term* is non-optional but may appear more than once.
- <*term*> is to denote the body of one complete syntax *term*.

The syntax of a rule definition is:

```
<rule> :==
(Define-Rule
<name>
<Documentation string>
<Antecendent body>
{Filter }*
-->
{Consequence body}+)

<name> :== atom
```

<Documentation string> :== "string"

<antecedent body> :==
{predicate sentence pattern}$^{+}$ |
{Filter }*

<predicate sentence pattern> :==
(<predicate/n> $<p - term_0>$ ... $<p - term_n>$)

<p-term> :==
({<p-term>}*) |
<variable> |
<constant>

<constant> :==
<atom>

<variable> :==
<?atom>

<Filter> :==
(Filter {Lisp lambda body}$^{+}$)

<consequence body> :==
<consequence body> |
<application>

<application> :==
(Lisp Function Application) |
(TELL <KNO Name> <predicate sentence pattern>) |
(DELETE <KNO Name> <ground predicate sentence pattern>) |

The rule's antecedent body consists of the conjunctive set of all *predicate sentence pattern* terms and the set of defined *filters*. When each pattern is individually satisified and all variable bindings in all bindings are unifiable and consistent and all *filters* are satisified for this set of variable bindings then the rule is triggered and its consequence body can be evaluated when scheduled.

A *filter-test* is any disembodied lambda function which returns non-nil to indicate a passing conditional and nil for a failing conditional.

An application can be any valid Common Lisp function application which will be evaluated in the binding environment of the rule when the rule's consequence body is executed.

Prototype KNO is designed to be an extension of Common Lisp. Lisp has two modes of evaluation. Evaluation at compile time and evaluation at load time. Both of these evaluation modes are necessary in order to specify when the binding of a variable should be dereferenced. With the advent of a rule's binding environment, it is necessary to delimit a third evaluation mode, evaluation at rule consequence execution time. Evaluation of a form in the rule's environment and the dereferncing of all variable bindings in that form is signaled by: (**reval** *form*).

B.4 KNO Performance

Preliminary performance results are available for *Prototype KNO*. In the earliest stages of the implementation of KNO emphasis was placed on performance.

Bench-1 tests the amount of time to perform 1000 basic knowledge base updates and the associated overhead of pattern matching one predicate sentence pattern with one variable. The benchmark is initiated with the operation
(TELL AGENT (status-of chain-is 1))
with the following rule in the rule base of the KNO *AGENT*.

```
(Define-Rule
  'AGENT
  '(Chain-Forever-0
     (status-of chain-is ?n)
     (filter (< ?n 1000))
     -->
     (TELL AGENT  (status-of chain-is (reval (1+ ?n))))))
```

Bench-2 expands on the Bench-1 benchmark by measuring the performance of *Prototype KNO* when a predicate sentence is removed from the knowledge base. *DELETE* is the complementary operation to *TELL* and also results in a change of *AGENT*s knowledge base. The benchmark is initiated with the operation
(TELL AGENT (type-of chain-is 1))
with the following rule in the rule base of the KNO *AGENT*.

```
(Define-Rule
  'AGENT
  '(Chain-Forever-1
     (type-of chain-is ?n)
     (filter (< ?n 1000))
     -->
     (DELETE AGENT (type-of chain-is ?n))
     (TELL AGENT (type-of chain-is (reval (1+ ?n))))
     ))
```

Bench-3 tests the basic overhead of the satisfing conjunctive sentences and the unification of a shared variable. This diagnostic like Bench-1 and Bench-2 is performed over 1000 rule executions. The benchmark is initiated with the operation

```
(TELL AGENT  (unif-bind-type-of 'Hairy-Moths 1))
(TELL AGENT  (unif-bind 'Hairy-Moths))
```

with the following rule in the rule base of the KNO *AGENT*.

```
(Define-Rule
  'AGENT
  '(Chain-Forever-2
    (unif-bind-type-of  ?m ?n)
    (unif-bind ?m)
    (filter (< ?n 1000))
    -->
    (TELL AGENT (unif-bind-type-of ?m (reval (1+ ?n))))))
```

The performance evaluation of *Prototype KNO* was performed in the following hardware/software configurations:

Symbolics 3620
3 Megawords of memory
Using Symbolics Common Lisp Genera V7.1
Timer resolution: 1.0 microseconds

Gold Hill Common Lisp V3.0
AI Architects Hummingboard
1.5 Megawords of memory
Processor: Intel 80386 at 16 MHz
Timer resolution: 10,000 microseconds

The results of the benchmarks are shown in Table B.1. The times shown in all these tables are in seconds. The number given in each column is the average execution time in a set of three runs.

Allocating memory is an expensive operation in the Symbolics and Gold Hill environments. This is highlighted when comparing the results of Bench-1 and Bench-2. In Bench-2, a predicate sentence is deleted from the knowledge base on each cycle. Because KNO has its own internal facility for the reclaiming of its dynamic data structures, the amount of time to *DELETE* a predicate sentence is more than made up by the fact that the *TELL* can use the reclaimed data structures instead of allocating them from the general Lisp environment memory.

Table B.1: Performance of Prototype KNO

Bench Diagnostic	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)	GoldHill Symbolics Ratio
Bench-1	3.8	19	5
Bench-2	2.5	21	8
Bench-3	7	38	5

B.5 Installing and Loading KNO

1. The KNO package uses the same DEFSYS.LSP utility as the CLBENCH package. The steps outlined in *Loading the CLBENCH* are identical to those that need be done here.

2. Modify the KNOSYS.LSP file in the same manner as outlined for the KNOSYS.LSP file for the CLBENCH package.

3. Compile-load or evaluate KNOSYS.LSP

4. Invoke the function *Compile-KNO* to compile the package.

5. Invoke the function *Load-KNO* to load the package.

B.6 KNO Example

In this section a small session with the KNO development environment is reviewed. This exercise serves as a quick "cookbook" introduction to the standard rule based representation and processing capabilities of the *Prototype KNO Version.*

The commands typed by the user are delimited by Kno> at the start of a line. Any line that does not begin with this delimiter is the output resulting from the execution of the command.

```
Kno> (setf *print-circle* 't)
T
```

By setting this Common Lisp environment control variable to non-nil most infinite structures will be printed properly. KNO has many structures which mutally refer to each other. Setting this variable *usually* prevents an infinite dump to your terminal screen. This behavior is Common Lisp implementation dependent.

```
Kno> (in-package 'kno)
#<Package KNO 37614005>
```

Currently, the interface of KNO is not exported. All references to KNO operations external to the package 'KNO must be explicitly referenced by preceding the operation with the package name 'KNO.

```
Kno> (Define-Rule
  'AGENT2
  '(Chain-Forever-0
     (status-of chain-is ?n)
     (filter (< ?n 1000))
     -->
     (TELL AGENT2 (status-of chain-is (reval (1+ ?n))))))

Warning:
Undefined Knowledge object: AGENT2
KNO must be defined before rule definition.
For rule: (CHAIN-FOREVER-0 (STATUS-OF CHAIN-IS ?N) (FILTER (< ?N 1000)) -->
          (TELL AGENT2 (STATUS-OF CHAIN-IS (REVAL (1+ ?N)))))
NIL
```

This shows an example of the error checking that is available in the KNO package. In this case an error resulted because a rule was defined for a KNO object instance *AGENT2* which was not defined.

```
Kno> (create-kno 'agent2)
#S(KNO :NAME AGENT2
       :PREDICATE-WM NIL
       :RULE-WM NIL
       :KNO-WM NIL  ...)

Kno> (describe-kb 'agent2)
AGENT has an EMPTY Knowledge Base
NIL

Kno> (describe-rule-base 'agent2)
AGENT has an EMPTY Rule Knowledge Base
NIL
```

When a KNO is created it comes into being totally ignorant about the universe in which it exists.

```
Kno> (Define-Predicate 'AGENT2 'status-of 2)
#S(PREDICATE :PRINT-VALUE STATUS-OF
             :KNO AGENT2
             :CLASS PREDICATE
             :ARITY 3
             :M-LINK-DISPATCH-VECTOR NIL
             :M-LINK-TERMINAL NIL
             :GPC-TABLE NIL
             :TELL GENERIC-TELL
             :ASK GENERIC-ASK
             :DELETE GENERIC-DELETE
             :RETRACT GENERIC-RETRACT
             :EXPLAIN GENERIC-EXPLAIN
             :DESCRIBE GENERIC-DESCRIBE)
```

The above gives us a peek at how the dispatching mechanism for the *KNOOP* is specified by the type of the knowledge being processed. Each predicate defined for a *KNO* is a structure which contains a set of pointers, one pointer for each *KNO Operation.* In this case, for the predicate *Status-of*, the *KNOOPs TELL* and *DELETE* will invoke the default *KNOIMPs GENERIC-TELL* and *GENERIC-DELETE* respectively.

```
Kno> (gc-statistics)
*ENVIRONMENT-HEAP*              Length: 0
          Allocate count:   0
          Deallocate count: 0
*BINDING-HEAP*                  Length: 0
```

```
          Allocate count:   0
          Deallocate count: 0
*GPC-HEAP*                     Length: 0
          Allocate count:   0
          Deallocate count: 0
*F-LINK-HEAP*                  Length: 0
          Allocate count:   0
          Deallocate count: 0
*RULE-HEAP*                    Length: 0
          Allocate count:   0
          Deallocate count: 0
*JOIN-HEAP*                    Length: 0
          Allocate count:   0
          Deallocate count: 0
*M-LINK-HEAP*                  Length: 0
          Allocate count:   0
          Deallocate count: 0
*EQUAL-HASH-TABLE-HEAP*        Length: 0
          Allocate count:   0
          Deallocate count: 0
*EQ-HASH-TABLE-HEAP*           Length: 0
          Allocate count:   0
          Deallocate count: 0
*SVECTOR-HEAP*                 Length: 0
          Allocate count:   0
          Deallocate count: 0
*VECTOR-HEAP*                  Length: 0
          Allocate count:   0
          Deallocate count: 0
NIL


Kno> (Define-Rule
   'AGENT2
   '(Chain-Forever-0
      (status-of chain-is ?n)
      (filter (< ?n 1000))
      -->
      (TELL AGENT2 (status-of chain-is (reval (1+ ?n))))))

#S(RULE :NAME CHAIN-FOREVER-0
        :ID #:RULE5313
        :KNO
          #S(KNO :NAME AGENT2
                 :PREDICATE-WM ... )
```

```
Kno> (gc-statistics)
*F-LINK-HEAP*                     Length: 0
          Allocate count:   1
          Deallocate count: 0
*RULE-HEAP*                       Length: 0
          Allocate count:   1
          Deallocate count: 0
*JOIN-HEAP*                       Length: 0
          Allocate count:   0
          Deallocate count: 0
*M-LINK-HEAP*                     Length: 0
          Allocate count:   2
          Deallocate count: 0
*EQUAL-HASH-TABLE-HEAP*           Length: 0
          Allocate count:   0
          Deallocate count: 0
*EQ-HASH-TABLE-HEAP*              Length: 5
          Allocate count:   13
          Deallocate count: 10
*VECTOR-HEAP*                     Length: 8
          Allocate count:   29
          Deallocate count: 26
NIL
```

By displaying the KNO garbage collection statistics before and after a rule has been created, it is seen that the garbage collector tracks the data structures that implement a rule.

```
Kno> (describe-predicate-base 'agent2 :brief nil)

This structure is a <PREDICATE>
Name:                STATUS-OF
Owning KNO:          AGENT2
GPC Count:           NIL
NIL

Kno> (describe-rule-base 'agent2 :brief nil)
This structure is a <RULE>
Name:                CHAIN-FOREVER-0
id:                  RULE5313
Owning KNO:          AGENT2
Reference Count:     0
Forward Feeding Node: M-LINK5311
Original form:
(CHAIN-FOREVER-0 (STATUS-OF CHAIN-IS ?N) (FILTER (< ?N 1000)) -->
 (TELL AGENT2 (STATUS-OF CHAIN-IS (REVAL (1+ ?N)))))
```

```
RHS Lambda form:
(DEFUN CHAIN-FOREVER-0-RHS-5312 (BINDING)
  NIL
  (LET* (($KNO$-RULE-BINDING (RULE-BINDING (DEFINED-KNO? 'AGENT2)))
         ($KNO$-LEFT-ENVIRONMENT-0
           (ENVIRONMENT-SLOTS (BINDING-ENVIRONMENT BINDING)))
         (?N (SVREF $KNO$-LEFT-ENVIRONMENT-0 0)))
    (GENERIC-TELL 'AGENT2
                  'AGENT2
                  'STATUS-OF
                  (LIST 'CHAIN-IS (1+ ?N))
                  $KNO$-RULE-BINDING
                  ':INFERRED)))
NIL
```

The above displays some of capabilities of the *DESCRIBE* facilities.

```
Kno> (TELL 'AGENT2 '(STATUS-OF CHAIN-IS 1))
999
```

Currently, the protocol of *TELL* is to return the number of forward inferences that occured due to the assertion of new knowledge.

```
Kno> (DESCRIBE-STATISTICS 'AGENT2)
TELL processing statistics for KnowledgeObject     : AGENT2

Clause Element Matching
 Successful Element-matches                        : 1000
 Failed Element-matches                            : 1000
 Total Element-matches                             : 2000

Entire Pattern of Clause Matched
 Successful Matches                                : 999
 Failed Matches                                    : 1
 Total Matches                                     : 1000

Clause - Clause Unification
 Successful Merges                                 : 0
 Failed Merges                                     : 0
 Total Merges                                      : 0

 Rules Triggered                                   : 999
 Rules Fired                                       : 999
```

```
NIL

Kno> (GC-STATISTICS)
*ENVIRONMENT-HEAP*              Length: 2
          Allocate count:   1000
          Deallocate count: 1000
*BINDING-HEAP*                  Length: 2
          Allocate count:   999
          Deallocate count: 999
*GPC-HEAP*                      Length: 0
          Allocate count:   1000
          Deallocate count: 0
.
.
*SVECTOR-HEAP*                  Length: 2
          Allocate count:   1000
          Deallocate count: 1000
.
.
NIL

Kno> (DESCRIBE-KB 'AGENT2)
PREDICATE STATUS-OF             1000
(STATUS-OF CHAIN-IS 639)
.
.
.
(STATUS-OF CHAIN-IS 367)
```

Using various *Describe* functions the effects of the knowledge processing that occured in the KNO *AGENT2* can be examined. For example, **GPC-HEAP** statistic shows the number of *ground predicate clause sentences* that were allocated. An *Environment* is a data structure that contains the variable bindings of a *ground predicate sentence pattern*, while a *Binding* is the data structure that records the entire binding environment of a *ground predicate sentence pattern.*

```
Kno> (CLEAR-KB 'AGENT2)
NIL

Kno> (GC-STATISTICS)
*ENVIRONMENT-HEAP*              Length: 2
          Allocate count:   1000
          Deallocate count: 1000
*BINDING-HEAP*                  Length: 2
          Allocate count:   999
          Deallocate count: 999
*GPC-HEAP*                        Length: 1000
          Allocate count:   1000
          Deallocate count: 1000
.
.
.
```

By clearing the knowledge base of *AGENT2* and then examining the KNO environment garbage collection statistics we see that the data structures associated with the deleted *predicate sentences* have been reclaimed.

The following session shows the defining and processing of a rule that uses both the *TELL* and *DELETE* operations.

```
Kno> (create-kno 'AGENT)
#S(KNO :NAME AGENT
       :PREDICATE-WM NIL
       :RULE-WM NIL
       :KNO-WM NIL  ...)

Kno> (Define-Predicate 'AGENT 'type-of 2)
#S(PREDICATE :PRINT-VALUE TYPE-OF
             :KNO AGENT
             :CLASS PREDICATE
             :ARITY 3  ...)

Kno> (Define-Rule
   'AGENT
   '(Chain-Forever-1
     (type-of chain-is ?n)
     (filter (< ?n 1000))
     -->
     (DELETE AGENT (type-of chain-is ?n))
     (TELL AGENT (type-of chain-is (reval (1+ ?n))))
     ))

#S(RULE :NAME CHAIN-FOREVER-1
        :ID #:RULE5330
        :KNO
          #S(KNO :NAME AGENT  ... )

Kno> (GC-STATISTICS)
*ENVIRONMENT-HEAP*                Length: 2
          Allocate count:   1000
          Deallocate count: 1000
*BINDING-HEAP*                    Length: 2
          Allocate count:   999
          Deallocate count: 999
*GPC-HEAP*                        Length: 1000
          Allocate count:   1000
          Deallocate count: 1000
*SVECTOR-HEAP*                    Length: 2
          Allocate count:   1000
          Deallocate count: 1000
NIL

Kno> (TELL 'AGENT2 '(TYPE-OF CHAIN-IS 1))
Unknown predicate <<TYPE-OF>> for KNO <AGENT2>
TELL operation ignored
For pclf: (TYPE-OF CHAIN-IS 1)
NIL

Kno> (TELL 'AGENT '(TYPE-OF CHAIN-IS 1))
999

Kno> (DESCRIBE-STATISTICS 'AGENT)
TELL processing statistics for KnowledgeObject     : AGENT

Clause Element Matching
 Successful Element-matches                        : 1000
 Failed Element-matches                            : 1000
 Total Element-matches                             : 2000

Entire Pattern of Clause Matched
 Successful Matches                                : 999
 Failed Matches                                    : 1
 Total Matches                                     : 1000

Clause - Clause Unification
 Successful Merges                                 : 0
 Failed Merges                                     : 0
 Total Merges                                      : 0

 Rules Triggered                                   : 999
 Rules Fired                                       : 999
NIL

Kno> (GC-STATISTICS)
*ENVIRONMENT-HEAP*                Length: 2
```

```
          Allocate count:   2000
          Deallocate count: 2000
*BINDING-HEAP*                 Length: 2
          Allocate count:   1998
          Deallocate count: 1998
*GPC-HEAP*                     Length: 999
          Allocate count:   2000
          Deallocate count: 1999
*SVECTOR-HEAP*                 Length: 2
          Allocate count:   2000
          Deallocate count: 2000
NIL

Kno> (DESCRIBE-RULE-BASE 'AGENT)
RULE     CHAIN-FOREVER-1     999
NIL

Kno> (DESCRIBE-KB 'AGENT)
PREDICATE TYPE-OF              1
(TYPE-OF CHAIN-IS 1000)
NIL

Kno> (DESCRIBE-PREDICATE-BASE 'AGENT :BRIEF NIL)

This structure is a <PREDICATE>
Name:                STATUS-OF
Owning KNO:          AGENT
GPC Count:           NIL

This structure is a <PREDICATE>
Name:                TYPE-OF
Owning KNO:          AGENT
GPC Count:           1
NIL
```

The following shows one *KNO*, *AGENT*, can communicate with another *KNO*, *MANAGER*. It also demonstrates that the rule compiler treates the consequence body as an extended version of Common Lisp. Any CommonLisp function application can be embedded in the consequence body of a rule.

```
Kno> (Define-Rule
   'AGENT
   '(Chain-Forever-1
      (type-of chain-is ?n)
      (filter (< ?n 1000))
```

```
    -->
    (when (equal ?n 999)
          (TELL MANAGER (finished AGENT processing)))
    (DELETE AGENT (type-of chain-is ?n))
    (TELL AGENT (type-of chain-is (reval (1+ ?n))))
    ))
```

The following session shows the definition and knowledge processing of a rule with conjunctive predicate sentence patterns.

```
Kno> (Define-Predicate 'AGENT 'unif-bind 1)
#S(PREDICATE :PRINT-VALUE UNIF-BIND
             :KNO AGENT
             :CLASS PREDICATE
             :ARITY 2 ...)

Kno> (Define-Predicate 'AGENT 'unif-bind-type-of 2)
#S(PREDICATE :PRINT-VALUE UNIF-BIND-TYPE-OF
             :KNO AGENT
             :CLASS PREDICATE
             :ARITY 3  ...)

Kno> (DESCRIBE-KB 'AGENT)
PREDICATE STATUS-OF            NIL
PREDICATE TYPE-OF              1
(TYPE-OF CHAIN-IS 1000)
PREDICATE UNIF-BIND            NIL
PREDICATE UNIF-BIND-TYPE-OF    NIL
NIL

Kno> (Define-Rule
  'AGENT
  '(Chain-Forever-2
    (unif-bind-type-of  ?m ?n)
    (unif-bind ?m)
    (filter (< ?n 1000))
    -->
    (TELL AGENT (unif-bind-type-of ?m (reval (1+ ?n))))))
#S(RULE :NAME CHAIN-FOREVER-2
        :ID #:RULE5365
        :KNO
          #S(KNO :NAME AGENT ... )

Kno> (DESCRIBE-RULE-BASE 'AGENT)
RULE      CHAIN-FOREVER-0     0
RULE     CHAIN-FOREVER-1    999
RULE     CHAIN-FOREVER-2    0
NIL


Kno> (DESCRIBE-RULE-BASE 'AGENT :BRIEF NIL)

This structure is a <RULE>
Name:                  CHAIN-FOREVER-0
id:                    RULE5276
Owning KNO:            AGENT
Reference Count:       0
Forward Feeding Node: M-LINK5274
Original form:
(CHAIN-FOREVER-0 (STATUS-OF CHAIN-IS ?N) (FILTER (< ?N 1000)) -->
 (TELL AGENT (STATUS-OF CHAIN-IS (REVAL (1+ ?N)))))
RHS Lambda form:
(DEFUN CHAIN-FOREVER-0-RHS-5275 (BINDING)
  NIL
  (LET* (($KNO$-RULE-BINDING (RULE-BINDING (DEFINED-KNO? 'AGENT)))
         ($KNO$-LEFT-ENVIRONMENT-0
           (ENVIRONMENT-SLOTS (BINDING-ENVIRONMENT BINDING)))
         (?N (SVREF $KNO$-LEFT-ENVIRONMENT-0 0)))
    (GENERIC-TELL 'AGENT
                  'AGENT
                  'STATUS-OF
                  (LIST 'CHAIN-IS (1+ ?N))
                  $KNO$-RULE-BINDING
                  ':INFERRED)))

This structure is a <RULE>
Name:                  CHAIN-FOREVER-1
id:                    RULE5330
Owning KNO:            AGENT
Reference Count:       999
Forward Feeding Node: M-LINK5328
Original form:
(CHAIN-FOREVER-1 (TYPE-OF CHAIN-IS ?N) (FILTER (< ?N 1000)) -->
 (DELETE AGENT (TYPE-OF CHAIN-IS ?N))
 (TELL AGENT (TYPE-OF CHAIN-IS (REVAL (1+ ?N)))))
RHS Lambda form:
(DEFUN CHAIN-FOREVER-1-RHS-5329 (BINDING)
  NIL ...)))

This structure is a <RULE>
Name:                  CHAIN-FOREVER-2
```

```
id:                    RULE5365
Owning KNO:            AGENT
Reference Count:       0
Forward Feeding Node: JOIN5348
Original form:
(CHAIN-FOREVER-2 (UNIF-BIND-TYPE-OF ?M ?N)
                 (UNIF-BIND ?M)
                 (FILTER (< ?N 1000))
  -->
 (TELL AGENT (UNIF-BIND-TYPE-OF ?M (REVAL (1+ ?N)))))
RHS Lambda form:
(DEFUN CHAIN-FOREVER-2-RHS-5347 (BINDING)
  NIL ...)

Kno> (GC-STATISTICS)
*ENVIRONMENT-HEAP*             Length: 2
          Allocate count:   2000
          Deallocate count: 2000
*BINDING-HEAP*                 Length: 2
          Allocate count:   1998
          Deallocate count: 1998
*GPC-HEAP*                     Length: 999
          Allocate count:   2000
          Deallocate count: 1999
*SVECTOR-HEAP*                 Length: 2
          Allocate count:   2000
          Deallocate count: 2000
NIL


Kno> (DESCRIBE-STATISTICS 'AGENT)
TELL processing statistics for KnowledgeObject    : AGENT

Clause Element Matching
 Successful Element-matches                       : 1000
 Failed Element-matches                           : 1000
 Total Element-matches                            : 2000

Entire Pattern of Clause Matched
 Successful Matches                               : 999
 Failed Matches                                   : 1
 Total Matches                                    : 1000

Clause - Clause Unification
 Successful Merges                                : 0
 Failed Merges                                    : 0
 Total Merges                                        : 0

 Rules Triggered                                     : 999
 Rules Fired                                         : 999
NIL

Kno> (TELL 'AGENT '(UNIF-BIND-TYPE 'HAIRY-MOTHS 1))
Unknown predicate <<UNIF-BIND-TYPE>> for KNO <AGENT>
TELL operation ignored
For pclf: (UNIF-BIND-TYPE 'HAIRY-MOTHS 1)
NIL

Kno> (TELL 'AGENT '(UNIF-BIND-TYPE-OF 'HAIRY-MOTHS 1))
0


Kno> (DESCRIBE-KB 'AGENT)
PREDICATE STATUS-OF            NIL
PREDICATE TYPE-OF              1
(TYPE-OF CHAIN-IS 1000)
PREDICATE UNIF-BIND            NIL
PREDICATE UNIF-BIND-TYPE-OF    1
(UNIF-BIND-TYPE-OF 'HAIRY-MOTHS 1)
NIL

Kno> (TELL 'AGENT '(UNIF-BIND 'HAIRY-MOTHS))
999

Kno> (DESCRIBE-STATISTICS 'AGENT)
TELL processing statistics for KnowledgeObject     : AGENT

Clause Element Matching
 Successful Element-matches                        : 1000
 Failed Element-matches                            : 2001
 Total Element-matches                             : 3001

Entire Pattern of Clause Matched
 Successful Matches                                : 1999
 Failed Matches                                    : 2
 Total Matches                                     : 2001

Clause - Clause Unification
 Successful Merges                                 : 999
 Failed Merges                                     : 0
 Total Merges                                      : 999
 Rules Triggered                                          : 1998
 Rules Fired                                              : 1998
NIL

Kno> (DESCRIBE-KB 'AGENT)
PREDICATE STATUS-OF          NIL
PREDICATE TYPE-OF            1
(TYPE-OF CHAIN-IS 1000)
PREDICATE UNIF-BIND          1
(UNIF-BIND 'HAIRY-MOTHS)
PREDICATE UNIF-BIND-TYPE-OF   1000
(UNIF-BIND-TYPE-OF 'HAIRY-MOTHS 62)
(UNIF-BIND-TYPE-OF 'HAIRY-MOTHS 351)
(UNIF-BIND-TYPE-OF 'HAIRY-MOTHS 552)
(UNIF-BIND-TYPE-OF 'HAIRY-MOTHS 977)
.
.
.
(UNIF-BIND-TYPE-OF 'HAIRY-MOTHS 618)


Kno> (GC-STATISTICS)
*ENVIRONMENT-HEAP*              Length: 2
          Allocate count:   4000
          Deallocate count: 3000
*BINDING-HEAP*                  Length: 2
          Allocate count:   3997
          Deallocate count: 2997
*GPC-HEAP*                      Length: 0
          Allocate count:   3001
          Deallocate count: 1999
*SVECTOR-HEAP*                  Length: 3
          Allocate count:   4000
          Deallocate count: 3000
*VECTOR-HEAP*                   Length: 0
          Allocate count:   1119
          Deallocate count: 105
NIL

Kno> (CLEAR-KB 'AGENT)
NIL

Kno> (GC-STATISTICS)
*ENVIRONMENT-HEAP*              Length: 1002
          Allocate count:   4000
          Deallocate count: 4000
```

```
*BINDING-HEAP*                    Length: 1002
          Allocate count:   3997
          Deallocate count: 3997
*GPC-HEAP*                        Length: 1002
          Allocate count:   3001
          Deallocate count: 3001
*SVECTOR-HEAP*                    Length: 1003
          Allocate count:   4000
          Deallocate count: 4000
NIL

Kno>
```

Appendix C

Common Lisp Diagnostic and Bench Package

This chapter outlines the installation procedure for the *Common Lisp Diagnostic and Bench Package*. It also briefly describes how to run the package.

C.1 Loading the CLBENCH

1. The *CLBENCH* package uses the *Define-System* utility. This utility is contained in the file *DEFSYS.LSP* in the distribution kit. Copy DEFSYS.LSP to a chosen subdirectory.

2. Compile load DEFSYS.LSP. The *Define-System* utility is now ready for use by the *CLBENCH*, *GBENCH* and *PCL* packages.

3. Copy the sources of the *CLBENCH* package to a chosen directory.

4. The file for defining the *CLBENCH* system will automatically load the *Define-System* utility if it has not been already loaded. In order for the utility to be loaded the pathname for *Define-System* files must be known. Change the appropreiate pathname in the statement at the top of DEFBENCH.LSP.

   ```
   (eval-when (load eval)
     (unless (boundp '*SYSTEM-BUILDER-UTILITIES*)
       (load
         #+Symbolics "p19:>fbi>kno>defsys.bin"
         #+:gclisp "c:\\ghsrc\\defsys\\defsys.f2s"
         )
       (setf *SYSTEM-BUILDER-UTILITIES* :loaded)))
   ```

   For example if you have copied *DEFSYS.LSP* in the directory
   `C:\\Gold\\defsys\\`
   then change
   `#+:gclisp "c:\\ghsrc\\defsys\\defsys.f2s"`
   to
   `#+:gclisp "c:\\Gold\\defsys\\defsys.f2s"`.

5. While still in the file DEFBENCH.LSP, edit the variable `*cl-bench-source-directory*` to reflect the directory where the *CLBENCH* package files reside. For example if the *CLBENCH* package files are in the directory `/Lucid/clbench` and your environment is Lucid Lisp on an Unix workstation then change

```
(defvar *cl-bench-source-directory*
  #+Lucid                    (pathname "/usr/lucid/clbench")
  .
  .
  .
  )
```

to

```
(defvar *cl-bench-source-directory*
  #+Lucid                    (pathname "/Lucid/clbench")
  .
  .
  .
  )
```

6. Compile-load or evaluate DEFBENCH.LSP.
7. Invoke the function *Compile-CL-Bench* to compile the package.
8. Invoke the function *Load-CL-Bench* to load the package.

C.2 Running the CLBENCH

The variable **all-timers** will contain all the symbol names of all individual Lisp operations tested by CLBENCH. For example the test of the function *caddr* is referenced by the symbol name *CALL-CADDR*.

The entire set of tests can be run and the results logged to file by the invoking the function:
Run-Series-To-File *log-file-pathname*.

Individual tests can be invoked by the function:
Run-One *test-symbol-name*.

Appendix D

Meter Operation Utility

The Meter Operation Utility is a function for interactively testing and evaluating (metering) the computational resources consumed by a Common Lisp form. The utility is written in CommonLisp and is been transported to three different environments: the Symbolics 3600 Common Lisp environment, the Gold Hill Common Lisp environment, and the Texas Instruments Explorer environment.

D.1 Installation of MeterOP

The Meter Operation Utility is contained in the file Meterop.lsp. To install the Meter Operation Utility compile and load this file into your target lisp environment. For convienience, the compiled file can be automatically loaded by including it in your world image or user initilization load file list.

D.2 Method of Use

The interface of the Meter Operation Utility is through the function:
User:MeterOp *number-of-iterations function-application*
where:

number-of-iterations is the number of times function-application will be invoked.

function-application is a Common Lisp function application of the form *(function arg-body)*.

For example:

```
*(defvar arr (make-array 10))
ARR
*(meterop 10000 (aref arr 4))
Test< 0>: Internal time (microsec): 129.0
Test< 1>: Internal time (microsec): 128.0
Test< 2>: Internal time (microsec): 129.0
<All values are in units of microseconds>
 Min: 128.0, Max: 129.0, Average: 128.66667, Sigma: 0.4759858
NIL
```

```
*(meterop 10000 (svref arr 4))
Test< 0>: Internal time (microsec): 2.0
Test< 1>: Internal time (microsec): 2.0
Test< 2>: Internal time (microsec): 2.0
<All values are in units of microseconds>
 Min: 2.0, Max: 2.0, Average: 2.0, Sigma: 0.0
NIL
*
```

which shows that the current implementation of the Gold Hill Common Lisp compiler will optimize array access forms if the accessor *svref* is used. If the more common accessor *aref* is used the compiler makes no attempt at optimization.

Note that in the above example an iteration sample of 10000 was used. This is necessary as the total time to execute *number-of-iterations* of *function-application* must exceed the resolution of the Common Lisp implementation dependent function *get-internal-run-time*. The Gold Hill Common Lisp implementation of *get-internal-run-time* has a resolution of 10000 microseconds while the Symbolics implementation has a resolution of 1 microsecond.

Appendix E

Performance and Evaluation of Common Lisp Environments

The performance and evaluation of Common Lisp primitives were performed using the *CLBENCH* and *METEROP* tools. Results from *METEROP* used a repeatation count of 10,000 for the Lisp Machine and 100,000 for Gold Hill Common Lisp. The tests were run in the following hardware/software configurations:

Symbolics 3620
3 Megawords of memory
Using Symbolics Common Lisp Genera V7.1
Timer resolution: 1.0 microseconds

Gold Hill Common Lisp V3.0
AI Architects Hummingboard
1.5 Megawords of memory
Processor: Intel 80386 at 16 MHz
Timer resolution: 10,000 microseconds

The results of the benchmarks are shown in Tables E.1-E.7. The times shown in all these tables are in microseconds. The number given in each column is the minimum execution time in a set of ten runs. The results of benchmarking Lisp primitives should be used only as crude diagnostics of an implementation. Results can vary dramatically depending on the function being tested and the arguments used.

Table E.1: Benchmarks of Common Lisp Function Invocation

CommonLisp Primitive	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
0 arguments	4.5	20.7	4.6
1 arguments	5.98	24.5	4.1
2 arguments	7.93	28.0	3.5

Table E.1 shows the results of function invocation. One of the most significant and basic performance differences between Symbolics 3620 and Gold Hill/Hummingboard platform is the intrinsic time required to invoke a function call. This factor alone accounts for most of the differences in performing the Gabriel benchmarks as reported in Appendix G.

Table E.2: Benchmarks of Common Lisp List Operations

CommonLisp Primitive	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
First	2.8	15*	5
second	4.4	15*	-
Fifth	9.7	25*	-
Cdr	3.4	15*	-
Member	4.9	25*	5
Cons	13.2	40*	3
List	23.5	75*	3
Append	21.8	250*	11
Rplaca	3.9	11.0*	2
Rplacd	3.9	14.0*	3

Because the Symbolics list machine has special "cdr" bits for the encoding of lists the results are uniform and monitonically increase with the length of the list. The GCLisp implementation of lists and the operation of lists cause significant variation of diagnostic results. All results shown in Table E.2 for Gold Hill are "starred" to indicate that these are average results.

The results of array operation tests are shown in Table E.3. The Symbolics compiler takes special care to identify 1-dimenisional and 2-dimensional array references. The Gold Hill compiler only optimizes *simple vector* references. We believe this is because *svref* is used by *Defstruct* accessors in the Gold Hill implementation. These results explain the large performance differences seen in Gabriel benchmarks that use array references.

It fairly clear from the results shown in Table E.6 for *String Operations* that the Gold Hill string operations have not been optimized. The *string* function is particular fast on the Symbolics as it is implemented in microcode.

Considerably higher performance is expected for numerical calculations with the availability of a Intel 80387 processor on the Hummingboard. At the time of these benchmarks, such an option was not available.

Table E.3: Benchmarks of Common Lisp Array Operations

CommonLisp Primitive	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
Svref	3.65	2.0	0.5
Aref, 1D	3.64	128	35
Aref, 2D	7.21	472	65
Aref, 3D	180.5	714	4
Aref, 4D	217.1	802	4
Make-Array	59.8	1200	20
Vector-Push	30.7	260	9
Vector-Push-Extend	138.5	1700	12
Vector-Pop	55.4	230	4

Table E.4: Benchmarks of Common Lisp Symbol Operations

CommonLisp Primitive	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
Boundp	4.1	4	1
Fboundp	3.4	38	11
Intern	168.9	122	0.7
Gensym	572	22800	40

Table E.5: Benchmarks of Common Lisp Hash Table Operations

CommonLisp Primitive	Symbolics 3620	GoldHill Hummingboard	GoldHill Symbolics
Sxhash	89.5	170	2
Gethash, eq	70.1	220	3
Gethash, equal	382.4	1800	5

Table E.6: Benchmarks of Common Lisp String Operations

CommonLisp Primitive	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
String	10.2	1594	160
String-Upcase	486.0	8820	18
Make-String	67.6	1630	24

Table E.7: Benchmarks of Common Lisp Number Operations

CommonLisp Primitive	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
+, Integer	3.7	10	3
-, Integer	3.7	10	3
*, Integer	8.9	35	4
/, Integer	14.2	110	7
+, Float	10.1	208	20
-, Float	11.4	210	20
*, Float	13.0	300	25
/, Float	18.9	1340	70
Cos, Float	203.0	13000	65
Sqrt, Float	111.8	19000	170
Abs, Float	13.0	110	8
+, Bignum	111.0	50	0.5
-, Bignum	104.0	50	0.5
*, Bignum	346	1200	4
/, Bignum	3490	16000	4

Appendix F

Gabriel Bench Package

This package contains the benchmarks as reported in *Performance and Evaluation of Lisp Systems by R. P. Gabriel.* The orginal version of this package was created at Symbolics, Inc. It was translated to Common Lisp by Charlie Hornig with the subsequent development of the timing tools by Charlie Hornig and Dan Weinreb. The package was released to the public domain where it was enhanced by Symbiotics with the inclusion of statisical data and the *Svref* based benchmarks.

F.1 Loading the GBENCH

1. The GBENCH package uses the same DEFSYS.LSP utility as the CLBENCH package. The steps outlined in *Loading the CLBENCH* are identical to those that need be done here.
2. Modify the DEFGBEN.LSP file in the same manner as outlined for the DEFBENCH.LSP file for the CLBENCH package.
3. Compile-load or evaluate DEFGBEN.LSP
4. Invoke the function *Compile-Gabriel-Bench* to compile the package.
5. Invoke the function *Load-Gabriel-Bench* to load the package.

F.2 Running the GBENCH

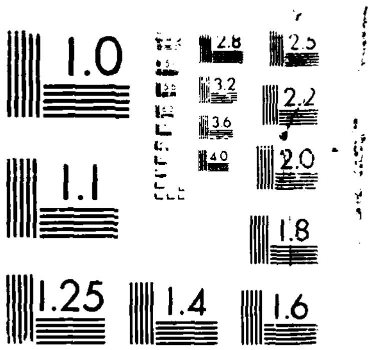
The variable **all-timers** will contain all the symbol names of all individual lisp operations tested by GBENCH. For example the test of the benchmark *Takr* is referenced by the symbol name *Takr*.

The entire set of tests can be run and the results logged to file by the invoking the function:
Run-Series-To-File *log-file-pathname*.

Individual tests can be invoked by the function:
Run-One *test-symbol-name*.

AD-A195 395 ACCESS: A COMMUNICATING AND COOPERATING EXPERT SYSTEMS 2/2
SYSTEM(U) SYMBIOTICS INC CAMBRIDGE MA
B H COTTMAN ET AL. 31 JAN 88 DAAB10-87-C-0053

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

Appendix G

Gabriel Benchmarks Performance and Evaluation

The Gabriel benchmarks were run on the following hardware/software configurations:

Symbolics 3620
3 Megawords of memory
Using Symbolics Common Lisp Genera V7.1
Timer resolution: 1.0 microseconds

Gold Hill Common Lisp V3.0
AI Architects Hummingboard
1.5 Megawords of memory
Processor: Intel 80386 at 16 MHz
Timer resolution: 10,000 microseconds

The results of the benchmarks are shown in Tables G.1-G.7. The times shown in all these tables are in seconds. The first number given in each column is the minimum execution time in the set of ten runs. The next number down is the average of running the benchmark ten times. The third number down, following the average time is the standard deviation of execution time of the set of ten measurements. The resolution of the available timing function available for the benchmarks varied from environment to environment.

Table G.1: Compilation Time of Gabriel Bench Package

Benchmark	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)
Compile	117	1910
	119	2120
	5	213

Table G.1 shows the amount of time required to compile the Gabriel benchmark file. This is a factor of approximately twenty. Two major effects are the cause of this slow down. The Symbolics has more memory and thus has to do less saving of compile state out to the disk. The second effect is due to the different architecture that the two compilers are targeting. In the case of the Symbolics compiler it targeting to a machine whose instruction set is designed for Lisp. Any Lisp compiler for the 80x86 family has a much more difficult task adapting Lisp to the underlying architecture's instruction set. The amount of instructions generated for the 80x86 is approximately four times that of Symbolics chip due to the extra instructions required for data type checking and 16-bit addressing of 32-bit words. However, these two factors do not explain this massive slowdown. An additional factor is that the Gold Hill compiler does not have a native 80286 or 80386 compiler. Instead, a 80286 backend compiler has been added as a pass to the old 8086 compiler. Obviously, the Gold Hill compiler could benefit from a complete overall. Improvements of factors of three in compiler speed are attainable.

Table G.2: Gabriel Benchmarks of Function Calling and Flow of Control

Benchmark	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)
Tak	0.449	2.42
	0.455	2.42
	0.004	0.00
STak	2.355	6.92
	2.380	6.92
	0.018	0.00
CTak	6.130	4.66
	6.166	4.67
	0.025	0.01
TakL	5.470	18.07
	5.488	18.11
	0.015	0.03
TakR	0.501	2.41
	0.507	2.43
	0.005	0.03

For function calling and list operations the Symbolics Common Lisp environment averaged about 5 times faster. The exceptional benchmark in this class was Browse. Browse makes heavy use of the function GENSYM for which the GCLisp on the Hummingboard is a factor of 50 times slower than the Symbolics. This resulted in Browse being executed almost 40 times faster on the Symbolics.

The benchmarks that used extensive array operations Puzzle and Triangle had a Symbolics-GoldHill-Hummingboard ratio of approximately 30 and 20 respectively. This is because direct 1-dimensional array references are on the order of 40 times slower using the current implementation of GoldHill Common Lisp. Also shown in Table G.4 is the effect of referencing arrays with *svref* instead of *aref*. For the benchmark *Triang* the improvement in performance is very significant.

Table G.3: Gabriel Benchmarks of List Manipulation

Benchmark	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)
Deriv	2.990	19.11
	3.005	19.11
	0.013	0.00
DDeriv	3.033	20.87
	3.041	20.91
	0.009	0.03
Div2,Iterative	1.517	4.34
	1.526	6.96
	0.010	1.85
Div2,Recursive	2.466	9.84
	2.484	9.91
	0.013	0.06
Destruct	1.930	13.67
	1.933	15.18
	0.002	2.09
Boyer	10.315	38.29
	10.325	41.53
	0.007	2.29
Browse	15.097	522.1
	15.128	525.7
	0.030	3.9
Trav-Init	6.786	63.2
	6.806	66.2
	0.023	4.0
Traverse	37.295	178.6
	37.303	178.6
	0.010	0.0

Table G.4: Gabriel Benchmarks of Array manipulation

Benchmark	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)
Puzzle	15.147	450.3
w/aref	15.158	451.8
	0.008	1.8
Puzzle	15.147	435.0
w/svref	15.158	435.1
	0.008	0.1
Triang	136.598	2605.2
w/aref	136.621	2605.3
	0.025	0.1
Triang	136.598	407.6
w/svref	136.621	408.0
	0.025	0.2

Table G.5: Gabriel Benchmarks of Numerical, Integer

Benchmark	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)
power=2	0.003	0.00
	0.004	0.02
	0.000	0.03
power=5	0.037	0.11
	0.038	0.12
	0.001	0.02
power=10	0.412	1.26
	0.415	1.28
	0.005	0.03
power=15	2.885	15.55
	2.894	15.69
	0.006	0.20

Table G.6: Gabriel Benchmarks of Numerical, Floating Point

Benchmark	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)
FFT	3.460	201.9
	3.489	207.2
	0.021	3.9
FFT w/svref	3.460	110.5
	3.489	110.5
	0.021	0.0
power=2	0.003	0.00
	0.004	0.03
	0.001	0.03
power=5	0.039	0.33
	0.041	0.33
	0.001	0.00
power=10	0.456	4.07
	0.463	4.07
	0.004	0.00
power=15	3.223	36.36
	3.250	36.51
	0.021	0.14

Table G.7: Gabriel Benchmarks of Numerical, Big Numbers

Benchmark	Symbolics 3620 (seconds)	GoldHill Hummingboard (seconds)
power=2	0.006	0.00
	0.007	0.03
	0.001	0.03
power=5	0.146	5.17
	0.148	6.78
	0.001	2.23
power=10	2.230	131.1
	2.248	131.6
	0.013	0.4
power=15	16.949	1404.0
	16.962	1437.2
	0.010	23.7

Appendix H

Optimized and Extended Portable Common Loops

H.1 Loading Optimized PCL

1. The OpPCL package uses the same DEFSYS.LSP utility as the CLBENCH package. The steps outlined in *Loading the CLBENCH* are identical to those that need be done here.
2. Modify the DEFOPPCL.LSP file in the same manner as outlined for the DEF-BENCH.LSP file for the CLBENCH package.
3. Compile-load or evaluate DEFOPPCL.LSP
4. Invoke the function *Compile-OpPCL* to compile the package.
5. Invoke the function *Load-OpPCL* to load the package.

H.2 OpPCL Extensions to the PCL Specification

The general reference to all the *PCL* and the *CLOS* specification in general is:
D.G. Bobrow and G. Kiczales.
Common Lisp Object System Specification
Draft X3 Document 87-003, Xerox PARC, 10-Feb-1987.
All work mentioned here is an extension of the implementation of PCL and as such retains the specified interface of *CLOS* with minor changes to avoid name conflicts with existing functional entry points.

defmethod-op *form*
defmethod-op is functionally equivalent to the PCL macro *defmethod*. *defmethod-op* determines the slot accessors at compile time by generating direct access forms for local slot values for all references to an object's slots within a method. The result is a significant speedup and a significant reduction in the amount of code generated. The disadvantage to the Symbiotics optimization is that all instances of a modified PCL object will not be updated. This is reconciled by recompiling all methods of an object when the object's definition is changed.

The following example shows the use of *defmethod-op* and *defmethod*. A method *access-op-5* is defined using *defmethod-op* which returns the values of slots *s1*, *s2*, *s3*, *s4* and *s5*. In a similar fashion the method *access-nm-5* is defined using *defmethod*.

```
(defclass pcl-obj () ((s1 :initform 1)
      (s2 :initform 2)
                          .
                          .
                          .
      (s30 :initform nil)
      )
    (:reader-prefix pcl-obj2-))

(defvar obj (make-instance 'pcl-obj))

(defmethod-op access-op-5 ((ob pcl-obj) x)
  (values x s1 s2 s3 s4 s5))

(defmethod access-nm-5 ((ob pcl-obj2) x)
  (with-slots (ob)
  (values x s1 s2 s3 s4 s5)))
```

A sense of the difference in the resulting implementation of the access methods generated by *defmethod-op* and *defmethod* can be achieved by applying *display-macro-expansion* or the equivalent on each of the forms. The resulting difference in code between these two forms is quite noticeable.

Appendix I

Optimized PCL Performance and Evaluation

In the early part of the Phase I effort, Symbiotics undertook the effort of improving Xerox PARC's public domain implementation of the developing Common Lisp Object System Specification [Bo87]. This package is commonly known as PCL or Portable Common Loops. PCL had been designed by the original architects to be a template with which a more optimized version could be built according to the target port environment. This appendix discusses the approach taken by Symbiotics to improve the performance of PCL and the results of those improvements.

I.1 Approach Taken To Improving PCL Performance

As the results of the performance study of PCL following this section will show, PCL suffers serious performance degradation over other object oriented packages. Our main objective in studying the design of PCL was to identify areas where performance could be improved. Also an equally important objective, was that any performance improvements gained did not come at the cost of changing the function specification of the CLOS interface presented.

One of the design decisions made in developing the original PCL was to support dynamic object definitions. In a problem domain where it is expected that descriptions change rapidly relative to the rate at which those descriptions are accessed, an interpreted inheritance scheme is usually computationally optimal. An interpreted inheritance scheme is implemented by retaining all descriptions locally to the defining parent object and chaining back through the inheritance chain to resolve any query for a description. The time required for retrieval in this approach is linearly proportional to the length of the inheritance chain. The upside of this approach is the ease of implementing and the computational efficiency of the description maintenance machinery. To change a description, only the description of the containing parent object need be modified. Using this implementation strategy, a description's representation is localized to one body, its value distributed only upon access. This approach has the disadvantage that accessing the state of the object is prohibitively expensive for dedicated applications in which object definitions do not change.

In our approach to improving the performance of PCL we looked for an alternative to that of determining object descriptions at runtime. Instead, we altered the way PCL creates object state accessors. In our approach object state locations are determined at compile time. Object state access is a direct reference into the structure of the object.

I.2 Results of PCL Performance Evaluation

The Symbiotics PCL benchmarks were run on the following hardware/software configurations:

Symbolics 3620
3 Megawords of memory
Using Symbolics Common Lisp Genera V7.1
Timer resolution: 1.0 microseconds

Gold Hill Common Lisp V3.0
AI Architects Hummingboard
1.5 Megawords of memory
Processor: Intel 80386 at 16 MHz
Timer resolution: 10,000 microseconds

The following tables summarize the results of running the various classes of Symbiotics PCL benchmarks. All times in Tables I.1 through I.5 are in microseconds. The first number given in each column is the average execution time averaged over one thousand trials. The next number down, following the average time is the standard deviation of execution time of the set of one thousand trials. The variable under test is the number of different slots accessed in a single method call. The resolution of the available timing function available for the benchmarks varied from environment to environment.

Table I.1 shows the performance results of original PCL as supported by Xerox PARC. The amount of time increases nonlinearly in proportion to the number of slot accessed. The locators of the most recent slots accessed are cached in a hash table of length 15. Any collisions in the cache causes the previous locator to be flushed in preference to the most recently calculated locator. If a locator is not in the cache then it must be calculated. The calculation of a slot locator is very expensive compared to a direct access. For the object used in this test, thrashing or a high collision rate 'n the cache does not occur until 10 slot accesses on the Symbolics. It occurs at 5 slot accesses on the GoldHill/Hummingboard experiment. This is no reflection on either environment but rather a matter of accident. The five or ten slots chosen just happen to thrash less on the Symbolics as their names when hashed do not collide as often. The hash function on the Symbolics is 32-bit wide while the GoldHill one is 16-bit wide. The 30-slot benchmark could not be compiled using the current version of GoldHill Common Lisp and 6 megabytes of memory. Compilation failure resulted due to running out of "atomspace".

Table I.2 shows the results of an experiment with generic PCL in which it was assured there was a 100% slot access cache hit rate. Slot names were chosen which did not collide with each other and thus cause cache thrashing for 1, 5 and 10 slot accesses. The PCL cache is 15 bins long and holds the locators of the last 15 slots accessed. If the locator is not residing in the cache then it is calculated. The method which has 30 slot accesses is guaranteed to cause this cache strategy to fail and thus measures the rate of slot access when the locator

Table I.1: Unoptimized PCL

Number of Slots Accessed	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
1	102	662	6.5
	1	50	–
5	156	3154	20.2
	2	77	–
10	518	7790	15.0
	4	53	–
30	9331	*	
	14	–	–

must be calculated after a collision.

Table I.2: Unoptimized PCL with 100% slot cache hit rate

Number of Slots Accessed	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
1	102	668	6.5
	1	41	–
5	157	1370	8.7
	3	37	–
10	224	2310	10.3
	3	50	–

Table I.3 shows the timing results of the Symbiotics optimized PCL. In this version slot locators are calculated at compile-time not run-time. The locators are placed in-line of the method being defined. The performance ratio we now see between GoldHill/Hummingboard and the Symbolics 3620 is predicted by the results of Common Lisp and Gabriel benchmark experiments. Optimized PCL methods are predominately function calls and direct structure access. The initial overhead of finding and dispatching the method call is approximately 90 μs on the Symbolics and 630 μs for GoldHill. With the optimizations, the execution time increases linearly with the number of slots accessed. This increase is approximately 2 μs for the Symbolics and 5 μs for GoldHill, the time of a slot access.

Table 4 summarizes the results by comparing the original implementation of PCL versus the optimized version for the GoldHill/Hummingboard environment. This comparsion clearly

Table I.3: Optimized PCL

Number of Slots Accessed	Symbolics 3620 (μs)	GoldHill Hummingboard (μs)	GoldHill Symbolics Ratio
1	91	628	6.9
	1	26	-
5	101	636	6.3
	1	43	-
10	112	662	5.9
	1	45	-
30	167	774	4.6
	1	28	-

shows that calculating the slot locator at runtime is the major consumer of cycles in a method call.

Table I.4: Optimized PCL vs. Unoptimized PCL for GoldHill/Hummingboard Environment

Number of Slots Accessed	Optimized PCL (μs)	Unoptimized PCL (μs)	Unoptimized Optimized Ratio
1	628	662	1.05
	26	50	-
5	636	3154	5.0
	43	77	-
10	662	7790	11.8
	45	53	-
30	774	22000	28.4
	28		-

Table I.5 summarizes the results by comparing the original implementation of PCL versus the optimized version for the Symbolics environment. Also, shown in Table I.5 is the timing results using the object oriented system available on Symbolics machines, Flavors. The intial cost of finding and dispatching a PCL method is much higher than dispatching a Flavor method. After this base cost is absorbed, however, we see that slot access of both Flavors and optimized PCL pay the same price of about 2 μs.

Table I.5: Optimized PCL vs. Unoptimized PCL for Symbolics/3620 Environment

Number of Slots Accessed	Optimized PCL (μs)	Unoptimized PCL (μs)	Flavors (μs)
1	91 1	102 1	21 1
5	101 1	156 2	28 1
10	112 1	514 4	38 1
30	167 1	9331 14	82 1

I.3 Summary of PCL Optimization Result

An unoptimized version of PCL in the Gold Hill/Hummingboard environment is not currently viable as a substrate on which to build any sizable or robust application. The exponential slowdown in slot access time with increasing number of different slot accesses is a considerable burden. Object oriented applications can easily have objects with slots numbering over 5 and often over 10. As the slot "caching" scheme of PCL is a hashtable of length 15, on the average slot index calculation thrashing becomes significant when 3 or more different slots are accessed during method calls. Also, indirect indexing code is generated at the site of each slot reference which results in an excessive amount of code generation. This effect is most apparent when we were not able to compile a PCL method that made 30 slot accesses. The code expansion of the method consumed all of memory in the GoldHill/Hummingboard and resulted in the crashing of the GCLisp environment.

The Symbiotics optimization of PCL was done by generating direct access forms for local slot values for all references to an object's slots within a method. The result is a significant speedup and a significant reduction in the amount of code generated. The disadvantage to the Symbiotics optimization is that all instances of a modified PCL object will not be updated. This is reconciled by rebinding the old instances of the old PCL object with the new definition of the PCL object. Future work would be the development of a utility that tracks all object definitons and their instances. When an object definition is changed the instance and all children of the object can be notified and recompiled with the new slot accessors.

Bibliography

[Ag82] Agerwala and Arvind *Data Flow Systems* Computer, Vol. 15, No. 2, Feb-1982

[Ag87] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems* M.I.T. Press, Cambridge MA, 1987.

[An86] F. Andre, D. Herman and J.P. Verjus *Synchronization of Parallel Programs* MIT Press, 1986

[Ba83] G. Barton. *A Multiple-Context Equality-Based Reasoning System* AI-TR-715, M.I.T. Artificial Intelligence Laboratory, April-1983.

[Ba86] M. Blanks. *Concurrent Cooperating Knowledge Bases* Science Applications International Corporation, McLean, VA. Oct-1986.

[Be86] G. Blelloch. *CIS: A Massively Concurrent Rule-Based System* M.I.T. Artificial Intelligence Memo-739, 1986.

[Bi73] G.M. Birtwistle *SIMULA begin* Auerbach Publishers, 1973.

[Bi87] R. Bisiani, F. Alleva, F. Correrini, A. Forin, F. Lecouat, and R. Lerner *Heterogeneous Parallel Processing: The Agora Shared Memory* Carnegie-Mellon University, Computer Science Department, CMU-CS-87-112. March 1987.

[Bo86] D.G. Bobrow et al. *CommonLoops: Merging Lisp and Object-Oriented Programming* Working paper for OOPSLA'86 Proceedings, 1986.

[Bo87] D.G. Bobrow and G. Kiczales. *Common Lisp Object System Specification* Draft X3 Document 87-003, Xerox PARC, 10-Feb-1987.

[Br75] Per Brinch Hansen *The Programming Language Concurrent Pascal* IEEE Trans. on Software Eng., Vol. SE-1, No. 2, June-1975.

[Br83] R. Brachman, R. Levesque and H. Levesque. *Krypton: A Functional Approach to Knowledge Representation* IEEE Computer,Vol. 16, No. 10, pp. 67-73, Oct-1983.

[CG87] Carnegie Group. *KnowledgeCraft CRL Technical Manual, Version 3.1* Carnegie Group 1987.

[Co86] B. J. Cox. Object-Oriented Programming: An Evolutionary Approach Addison-Wesley. Reading, Massachusetts. 1986.

[Cl82] D. D. Clark *Internet Protocol Implementation Guide* SRI International, Menlo Park, CA, August 1982

[Da81] R. Davis and R. Smith. *Negotiation as a Metaphor for Distributed Problem Solving* M.I.T. Artificial Intelligence Laboratory Memo-624, 1981.

[De87] B. A. Delagi. N. P. Saraiya, and G. T. Byrd *Lamina: Care Applications Interface* Stanford University, Knowledge Systems Laboratory, KSL 86-67. Nov. 1987

[Di68] E.W. Dijkstra *Co-operating Sequential Processes* Programming Languages - NATO Advanced Study Institute, Academic Press, pp. 43-110, 1968.

[Dk86] J. deKleer. *An Assumption-Based Truth Maintenance System* Artificial Intelligence Vol.28 pp. 127-162, 1986.

[Do79] J. Doyle. *A Truth Maintenance System* Artificial Intelligence Vol.12 pp. 231-272 1979.

[ED87] Nicolas Mokhoff *Five-chip token-passing set operates LANS at 100 Mbits/s* Electronic Design Vol. 35,No. 22 pp. 45-50, Sept-17,1985.

[Fa87] Joseph R. Falcone. *A Programmable Interface Language for Heterogenous Distributed Systems* ACM Transactions on Computer Systems, Vol. 5, No. 4, November 1987, pp. 330-351.

[Fi85] R. Fikes and T. Kehler. *Frame-Based Representation in Reasoning* Communications of the ACM Vol. 28(9) pp. 904-920.

[Fi85] R. Fikes and T. Kehler. *Frame-Based Representation in Reasoning* Communications of the ACM Vol. 28(9) pp. 904-920.

[Ga85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems* MIT Press, 1985.

[Ga86] L. Gasser, C. Braganza, and N. Herman. *MACE: A Flexible Testbed for Distributed AI Research* Distributed Artificial Intelligence Group, Computer Sci. Dept. USC, 9-Aug-1986.

[Ge82] M.R. Genesereth. *An Overview of Meta-Level Architecture* HPP-81-6, Heuristic Programming Project, Stanford University, Dec-1982.

[Ge87] M.R. Genesereth *Deliberate Agents* Technical Report Logic-87-1. Stanford University, Logic Group, 1987.

[Gr78] J.N. Grey *Notes on Database Operating Systems* Lecture Notes in Computer Science, Springer-Verlag, pp. 393-481, 1978.

[Gr80] R. Greiner. *RLL-1: A Representation Language Language* Working Paper 80-9, Heuristic Programming Project, Stanford University, Oct-1980.

[Ha86] B. Hayes-Roth. *A Blackboard Architecture for Control* Artificial Intelligence, Vol.262, pp.251-321, Mar-1986.

[Hi85] W. Daniel Hillis *The Connection Machine* MIT Press, 1985.

[Ho78] C.A.R. Hoare *Communicating Sequential Processes* CACM, Vol. 21, No. 8, pp. 666-677, Aug-1978.

[Hs86a] K. Hasse. *ARLO: Another Representation Language Offer* M.I.T. Artificial Intelligence Laboratory Technical Report 901, 1986.

[Hs86b] K. Hasse. *Why Representation Language Languages are No Good* M.I.T. Artificial Intelligence Laboratory Memo-943, 1986.

[HsS6c] K. Hasse. *Discovery Systems* M.I.T. Artificial Intelligence Laboratory Memo-899, Aug-1986.

[He77] C. Hewitt. *Viewing Control Structures as Patterns of Message Passing* Journal of Artificial Intelligence, pp. 323-364, June-1977.

[He85a] C. Hewitt, T.Reinhardt, G. Agha and G. Attardi. *Linguistic Support of Serializers For Shared Resources* In Seminar on Concurrency, p330-359, Springer-Verlag, 1985.

[He85b] C. Hewitt and P. deJong. *The Challenge of Open Systems* BYTE Vol. 10, pp.223-242, April-1985.

[He86] C. Hewitt. *Offices are Open Systems* ACM Transactions on Office Information Systems, Vol.4, No.3, p271-287, July-1986.

[Hi85] D. Hillis. *The Connection Machine* MIT Press, 1985.

[Inf87] *ART Reference Manual, Version 3.0* Inference Corporation, 1987.

[In85] *KEE Software Development System User's Manual, Version 3.0* IntelliCorp, 1985

[ISO79] ISO TC 97 SC16. *Open Systems Interconnection* N227 August, 1979.

[Kn68] D. Knuth. *Semantics of Context-Free Languages* Mathematical Systems Theory Vol. 39, pp. 127-145, 1968.

[Ko81] K. Konolige. *A First-Order Formalization of Knowledge and Action for a Multi-Agent Planning System* Machine Intelligence, Vol.10, pp.41-72, 1981.

[Kr81] W.A. Kornfeld and C. Hewitt. *The Scientific Community Metaphor.* M.I.T. Artificial Intelligence Memo-750, Jan-1981.

[KCS86] Knowledge Systems Corporation *Evaluation of AI Languages and Knowledge Engineering Environments* Knowledge Systems Corporation, 1986

[Le82] D. Lenat. *AM: Discovery In Mathematics as Heuristic Search* Knowledge Based Systems in Artificial Intelligence, McGraw Hill, 1982.

[Le83] D. Lenat. *Eurisko: A Program That Learns New Heuristics and Domain Concepts* The AI Journal, March-1983.

[Le86] D. P. O'Leary, G.W. Stewart and Robert van de Geijn. *DOMINO: A Message Passing Environment for Parallel Computation* University of Maryland, Computer Science Technical Report Series. TR-1648. April, 1986.

[Li79] Barbara Liskov *Primitives for Distributed Computing* Proc. Seventh ACM Symp. on Operating Systems, pp. 33-42, 1979.

[Lo86] Loh-Ping Yu *The Anatomy of a Distributed Electronic Mail Network* The Executive Guide to Data Communications, Vol. 8, pp. 104-107,

[Ly81] N. Lynch and J. Fischer *On Describing Behavior and Implementation of Distributed Systems* Theoretical Comp. Sci., Vol. 13, No. 1, 1981

[Ma85] C.R. Manning. *Organizing Sprites* M.I.T. Artificial Intelligence Laboratory Memo, 1985.

[Mc82] D. McAllester. *Reasoning Utility Package User's Manual* AI Memo 667, M.I.T. Artificial Intelligence Laboratory, Cambridge MA, 1982.

[Mi86] M. Minsky. *The Society of Mind* (Simon & Schuster, 1986

[Mo85] J. Elliott B. Moss *Nested Transactions - An Approach to Reliable Distributed Computing* MIT Press, 1985.

[Ne81] B.J. Nelson *Remote Procedure Call* Technical Report CSL-81-9, Xerox PARC, 1981.

[Pa87] Robert C. Paslay *Persistent Object Definitions - PODS* Symbiotics Inc., Cambridge, MA Dec-1987.

[Po85] Gerald Popek and Bruce J. Walker *The Locus Distributed System Architecture* MIT Press, 1985.

[Po87] Dick Pountain and David May *A Tutorial Introduction to OCCAM Programming* McGraw-Hill, 1987

[PP87] *Portable Programs for Parallel Processors* McGraw-Hill, 1987.

[Re82] T. Reps *Generating Language-Based Environments* Ph.D Thesis. Cornell University, Aug-1982.

[Ro86] John L. Romkey *PC-IP Programmer's Manual* Laboratory for Computer Science, MIT, April-1986.

[Ro87] S. Rowley, H. Shrobe, R. Cassels and W. Hamscher *Joshua: Uniform Access to Heterogenous Knowledge Sources* AAAI-87, pp. 48-52. Seattle, Wa

[Ru85] S. Russel *The Complete Guide to MRS* Stanford Knowledge Systems Laboratory Report No. KSL-85-12. Stanford, Ca. 1985.

[Ru87] D.E. Rumelhart, McCelland, J.L. and the PDP Research Group MIT Press, 1987

[Sa85] H.S.H. Sandell and R.W. Worrest. *Cooperative Real-Time Planning: An Experimental Study* 30th North Carolina IEEE Symposium and Exhibition, pp.79-84, Oct-1985.

[Sa86] Jerome H. Saltzer and John L. Romkey *PC/IP User's Guide* Laboratory for Computer Science, MIT, April-1986.

[Sc86] L. Schubert and F. Peiletier. *From English to Logic: Context-Free Computation of 'Conventional' Logical Translations* In Readings in Natural Language Processing ed. by B. Grosz, K. S. Jones, and B. Webber. Morgan Kaufmann Pub. Inc., Los Altos, Ca. 1986.